USENIX

conference

proceedings

2nd Symposium on Networked Systems Design & Implementation

Boston, MA, USA

May 2–4, 2005

Sponsored by

The USENIX Axssociation

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

in cooperation with ACM SIGCOMM and ACM SIGOPS

Proceedings of the 2nd Symposium on Networked Systems Design & Implementation

Boston, MA, USA, May 2005

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
URL: http://www.usenix.org

The price is $40 for members and $50 for nonmembers.
Outside the U.S.A. and Canada, please add
$20 per copy for postage (via air printed matter).

Past NSDI Symposium

First Symposium on Networked Systems Design and Implementation
March 29–31, 2004, San Francisco, CA, USA

© 2005 by The USENIX Association
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN 1-931971-30-7

USENIX Association

Proceedings of the 2nd Symposium on Networked Systems Design & Implementation (NSDI '05)

May 2–4, 2005
Boston, MA, USA

Conference Organizers

Program Chairs

Amin Vahdat, *University of California, San Diego*
David Wetherall, *University of Washington*

Program Committee

Miguel Castro, *Microsoft Research*
Jon Crowcroft, *University of Cambridge*
David Culler, *University of California, Berkeley*
Michael Dahlin, *University of Texas at Austin*
Peter Druschel, *Rice University*
Paul Francis, *Cornell University*
Ramesh Govindan, *University of Southern California*
Joe Hellerstein, *Intel Research and University of California, Berkeley*
Dina Katabi, *Massachusetts Institute of Technology*
Eddie Kohler, *University of California, Los Angeles*
Ed Lazowska, *University of Washington*
Jeffrey Mogul, *HP Labs*
Vern Paxson, *ICSI and Lawrence Berkeley National Laboratory*
Larry Peterson, *Princeton University*
Stefan Savage, *University of California, San Diego*
Srinivasan Seshan, *Carnegie Mellon University*
Ellen Zegura, *Georgia Institute of Technology*

Steering Committee

Thomas Anderson, *University of Washington*
Peter Honeyman, *CITI, University of Michigan*
Mike Jones, *Microsoft Research*
Robert Morris, *Massachusetts Institute of Technology*
Mike Schroeder, *Microsoft Research*
Amin Vahdat, *University of California, San Diego*
Ellie Young, *USENIX Association*

Poster Session Chairs

Eddie Kohler, *University of California, Los Angeles*
Michael Dahlin, *University of Texas at Austin*

The USENIX Association Staff

External Reviewers

Mukesh Agrawal
Aditya Akella
Nazrul Alam
David Andersen
Thomas Anderson
Remzi Arpaci-Dusseau
Mary Baker
Magdalena Balazinska
Hitesh Ballani
Andy Bavier
Brian Bershad
Ashwin Bharambe
Ken Birman
Mike Burrows
John Byers
Yatin Chawathe
Allen Clement
Richard Cox
Constantinos Dovrolis
Richard Dunn
Jeremy Elson
Cristian Estan
Nick Feamster
Marc Fiuczynski
Sally Floyd
Paul Gauthier
Steven Gribble
Timothy Griffin
Saikat Guha
Krishna Gummadi
Andreas Haeberlen
Jeff Hoye
Mark Huang
Jaeyeon Jung
Flavio Junqueira
Frans Kaashoek
Shrikanth Kandula
Scott Karlin
Sachin Katti
Dejan Kostic
David Kotz
Boon Thau Loo
David Lowenthal
Ratul Mahajan
Petros Maniatis
Tudor Marian
David Mazières
Alan Mislove
Michael Mitzenmacher
Steve Muir
Aki Nakao
Suman Nath
David Oppenheimer
Vivek Pai
Jeff Pang
Ruoming Pang
Kurt Partridge
Adrian Perrig
Ansley Post
Kostas Psounis
Venugopalan Ramasubramanian
Charles Reis
Maya Rodrig
Dan Sandler
Stefan Saroiu
Marianne Shaw
Alan Shieh
Atul Singh
Manpreet Singh
Alex Snoeren
Yee Jiun Song
Tammo Spalink
Neil Spring
Ion Stoica
Michael Swift
George Varghese
Vidhyashankar Venkataraman
Arun Venkataramani
Vivek Vishnumurthy
Geoffrey Voelker
Kevin Walsh
Matt Walsh
Andrew Whitaker
Dan Williams
Bernard Wong
Kenneth Yocum
Haifeng Yu
John Zahorjan
Ming Zhang
Xinyang Zhang

NSDI '05: 2nd Symposium on Networked Systems Design & Implementation

Boston, MA, USA

Index of Authors vii
Message from the Program Chairs ix

Monday, May 2, 2005

Internet Routing
Session Chair: Jon Crowcroft, University of Cambridge

Finding a Needle in a Haystack: Pinpointing Significant BGP Routing Changes in an IP Network 1
Jian Wu and Zhuoqing Morley Mao, University of Michigan; Jennifer Rexford, Princeton University; Jia Wang, AT&T Labs—Research

Design and Implementation of a Routing Control Platform 15
Matthew Caesar, University of California, Berkeley; Donald Caldwell, AT&T Labs—Research; Nick Feamster, Massachusetts Institute of Technology; Jennifer Rexford, Princeton University; Aman Shaikh and Jacobus van der Merwe, AT&T Labs—Research

Negotiation-Based Routing Between Neighboring ISPs 29
Ratul Mahajan, David Wetherall, and Thomas Anderson, University of Washington

Models and Faults
Session Chair: Amin Vahdat, University of California, San Diego

Detecting BGP Configuration Faults with Static Analysis 43
Nick Feamster and Hari Balakrishnan, MIT Computer Science and Artificial Intelligence Laboratory

IP Fault Localization Via Risk Modeling 57
Ramana Rao Kompella, University of California, San Diego; Jennifer Yates and Albert Greenberg, AT&T Labs—Research; Alex C. Snoeren, University of California, San Diego

Performance Modeling and System Management for Multi-component Online Services 71
Christopher Stewart and Kai Shen, University of Rochester

Overlays and DHTs
Session Chair: Peter Druschel, Rice University

Debunking Some Myths About Structured and Unstructured Overlays 85
Miguel Castro, Manuel Costa, and Antony Rowstron, Microsoft Research Cambridge

Bandwidth-efficient Management of DHT Routing Tables 99
Jinyang Li, Jeremy Stribling, Robert Morris, and M. Frans Kaashoek, MIT Computer Science and Artificial Intelligence Laboratory

Improving Web Availability for Clients with MONET 115
David G. Andersen, Carnegie Mellon University; Hari Balakrishnan, M. Frans Kaashoek, and Rohit N. Rao, MIT Computer Science and Artificial Intelligence Laboratory

Tuesday, May 3, 2005

Storage
Session Chair: Mike Dahlin, University of Texas, Austin

Shark: Scaling File Servers via Cooperative Caching 129
Siddhartha Annapureddy, Michael J. Freedman, and David Mazières, New York University

Glacier: Highly Durable, Decentralized Storage Despite Massive Correlated Failures 143
Andreas Haeberlen, Alan Mislove, and Peter Druschel, Rice University

Building Network Services
Session Chair: Larry Peterson, Princeton University

Quorum: Flexible Quality of Service for Internet Services 159
Josep M. Blanquer, Antoni Batchelli, Klaus Schauser, and Rich Wolski, University of California, Santa Barbara

Trickles: A Stateless Network Stack for Improved Scalability, Resilience, and Flexibility 175
Alan Shieh, Andrew C. Myers, and Emin Gün Sirer, Cornell University

Designing Extensible IP Router Software 189
Mark Handley, University College, London, and International Computer Science Institute; Eddie Kohler, University of California, Los Angeles, and International Computer Science Institute; Atanu Ghosh, Orion Hodson, and Pavlin Radoslavov, International Computer Science Institute

Wireless
Session Chair: Eddie Kohler, University of California, Los Angeles

Using Emulation to Understand and Improve Wireless Networks and Applications 203
Glenn Judd and Peter Steenkiste, Carnegie Mellon University

Geographic Routing Made Practical 217
Young-Jin Kim and Ramesh Govindan, University of Southern California; Brad Karp, Intel Research/Carnegie Mellon University; Scott Shenker, University of California, Berkeley/ICSI

Sustaining Cooperation in Multi-hop Wireless Networks 231
Ratul Mahajan, Maya Rodrig, David Wetherall, and John Zahorjan, University of Washington

System Management and Configuration
Session Chair: Jeff Mogul, Hewlett-Packard Labs

ACMS: The Akamai Configuration Management System 245
Alex Sherman, Akamai Technologies and Columbia University; Philip A. Lisiecki and Andy Berkheimer, Akamai Technologies; Joel Wein, Akamai Technologies and Polytechnic University

The Collective: A Cache-Based System Management Architecture 259
Ramesh Chandra, Nickolai Zeldovich, Constantine Sapuntzakis, and Monica S. Lam, Stanford University

Live Migration of Virtual Machines 273
Christopher Clark, Keir Fraser, and Steven Hand, University of Cambridge Computer Laboratory; Jacob Gorm Hansen and Eric Jul, University of Copenhagen; Christian Limpach, Ian Pratt, and Andrew Warfield, University of Cambridge Computer Laboratory

Wednesday, May 4, 2005

Security
Session Chair: Stefan Savage, University of California, San Diego

Botz-4-Sale: Surviving Organized DDoS Attacks That Mimic Flash Crowds 287
Srikanth Kandula and Dina Katabi, Massachusetts Institute of Technology; Matthias Jacob, Princeton University; Arthur Berger, Massachusetts Institute of Technology/Akamai

Cashmere: Resilient Anonymous Routing 301
Li Zhuang and Feng Zhou, University of California, Berkeley; Ben Y. Zhao, University of California, Santa Barbara; Antony Rowstron, Microsoft Research, UK

Sensor Networks
Session Chair: Srinivasan Seshan, Carnegie Mellon University

Decentralized, Adaptive Resource Allocation for Sensor Networks 315
Geoffrey Mainland, David C. Parkes, and Matt Welsh, Harvard University

Beacon Vector Routing: Scalable Point-to-Point Routing in Wireless Sensornets 329
Rodrigo Fonseca, University of California, Berkeley; Sylvia Ratnasamy, Intel Research, Berkeley; Jerry Zhao, International Computer Science Institute; Cheng Tien Ee and David Culler, University of California, Berkeley; Scott Shenker, University of California, Berkeley, and International Computer Science Institute; Ion Stoica, University of California, Berkeley

Active Sensor Networks 343
Philip Levis, University of California, Berkeley; David Gay, Intel Research Berkeley; David Culler, University of California, Berkeley

Index of Authors

Andersen, David G. 115
Anderson, Thomas 29
Annapureddy, Siddhartha 129
Balakrishnan, Hari 43, 115
Batchelli, Antoni 159
Berger, Arthur 287
Berkheimer, Andy 245
Blanquer, Josep M. 159
Caesar, Matthew 15
Caldwell, Donald 15
Castro, Miguel 85
Chandra, Ramesh 259
Clark, Christopher 273
Costa, Manuel 85
Culler, David 329, 343
Druschel, Peter 143
Ee, Cheng Tien 329
Feamster, Nick 15, 43
Fonseca, Rodrigo 329
Fraser, Keir 273
Freedman, Michael J. 129
Gay, David 343
Ghosh, Atanu 189
Govindan, Ramesh 217
Greenberg, Albert 57
Haeberlen, Andreas 143
Hand, Steven 273
Handley, Mark 189
Hansen, Jacob Gorm 273
Hodson, Orion 189
Jacob, Matthias 287
Judd, Glenn 203
Jul, Eric 273
Kaashoek, M. Frans 99, 115
Kandula, Srikanth 287
Karp, Brad 217
Katabi, Dina 287
Kim, Young-Jin 217
Kohler, Eddie 189
Kompella, Ramana Rao 57
Lam, Monica S. 259
Levis, Philip 343
Li, Jinyang 99
Limpach, Christian 273
Lisiecki, Philip A. 245
Mahajan, Ratul 29, 231
Mainland, Geoffrey 315
Mao, Zhuoqing Morley 1
Mazières, David 129
Mislove, Alan 143
Morris, Robert 99
Myers, Andrew C. 175
Parkes, David C. 315
Pratt, Ian 273
Radoslavov, Pavlin 189
Rao, Rohit N. 115
Ratnasamy, Sylvia 329
Rexford, Jennifer 1, 15
Rodrig, Maya 231
Rowstron, Antony 85, 301
Sapuntzakis, Constantine 259
Schauser, Klaus 159
Shaikh, Aman 15
Shen, Kai 71
Shenker, Scott 217, 329
Sherman, Alex 245
Shieh, Alan 175
Sirer, Emin Gün 175
Steenkiste, Peter 203
Stewart, Christopher 71
Stoica, Ion 329
Stribling, Jeremy 99
van der Merwe, Jacobus 15
Wang, Jia 1
Warfield, Andrew 273
Wein, Joel 245
Welsh, Matt 315
Wetherall, David 29, 231
Wolski, Rich 159
Wu, Jian 1
Yates, Jennifer 57
Zahorjan, John 231
Zeldovich, Nickolai 259
Zhao, Ben Y. 301
Zhao, Jerry 329
Zhou, Feng 301
Zhuang, Li 301

Message from the Program Chairs

We are very pleased to introduce these Proceedings of the 2nd Symposium on Networked Systems Design & Implementation (NSDI '05). It is an exciting time to be involved with NSDI. The conference continues to evolve and define its scope, following its successful inaugural year. NSDI was created to provide a high-quality venue for research and experience focused on the design and implementation of networked systems. We have taken a broad view of this charter, selecting papers from across the range of the SIGCOMM and SIGOPS communities, rather than their intersection. To provide a coherent scope for the conference, we favored systems-style work that takes place in a networking context. The result was a strong program with a broad set of papers, from wireless, through network filesystems and Internet routing, to network security. To complement these papers, the program also included a poster session that featured work in progress, and a keynote by Tom Leighton on Internet content delivery. It is our hope that NSDI will continue to bring together researchers and developers with expertise in traditionally disjoint domains for a productive interchange.

We received 112 paper submissions and subjected them to two rounds of review by both the program committee and outside experts. In the first round, each submission received four reviews. We used these reviews to select approximately half of the papers for a second round of review. A total of 568 reviews were written. We hope that this high level of feedback will help all authors improve the quality of their research and its presentation. The program committee then met in Seattle, Washington, in January 2005 and selected the 25 papers that appear in these proceedings. Each submission discussed at the PC meeting was read and personally reviewed by at least five PC members. This led to many lively and informed discussions! Finally, the accepted papers were individually shepherded by members of the program committee.

We would particularly like to thank the authors of submissions for trusting their work to a new conference. Ultimately, it is you, along with the conference participants, who will judge the success of NSDI and who will influence its future course. We are indebted to the members of the program committee, who personally contributed many hours of effort as they reviewed a large number of submissions. We especially thank the external reviewers, whose names are listed in the frontmatter. They contributed their time and expertise to make this a better conference. And we give special thanks to Ellie Young, Jane-Ellen Long, and Anne Dickison of USENIX, who gave us a course to steer by and ably handled the myriad details of organizing the conference.

We look forward to seeing you in Boston!

Amin Vahdat, University of California San Diego
David Wetherall, University of Washington
Program Chairs

Finding a Needle in a Haystack: Pinpointing Significant BGP Routing Changes in an IP Network

Jian Wu, Zhuoqing Morley Mao
University of Michigan

Jennifer Rexford
Princeton University

Jia Wang
AT&T Labs–Research

Abstract

The performance of a backbone network is vulnerable to *interdomain* routing changes that affect how traffic travels to destinations in other Autonomous Systems (ASes). Despite having poor visibility into these routing changes, operators often need to react quickly by tuning the network configuration to alleviate congestion or by notifying other ASes about serious reachability problems. Fortunately, operators can improve their visibility by monitoring the Border Gateway Protocol (BGP) decisions of the routers at the periphery of their AS. However, the volume of measurement data is very large and extracting the *important* information is challenging. In this paper, we present the design and evaluation of an online system that converts millions of BGP update messages a day into a few dozen *actionable* reports about significant routing disruptions. We apply our tool to two months of BGP and traffic data collected from a Tier-1 ISP backbone and discover several network problems previously unknown to the operators. Validation using other data sources confirms the accuracy of our algorithms and the tool's additional value in detecting routing disruptions.

1 Introduction

Ensuring good performance in an IP backbone network requires continuous monitoring to detect and diagnose problems, as well as quick responses from management systems and human operators to limit the effects on end users. Network operators need to know when destinations become unreachable to notify affected customers and track down the cause of the problem. When measurements indicate that links have become congested, operators may respond by modifying the routing protocol configurations to direct some traffic to other lightly-loaded paths. These kinds of measurements are also crucial for discovering weaknesses in existing network protocols, router implementations, and operational practices to drive improvements for the future. All of these tasks require effective ways to cull through large amounts of measurement data, often in real time, to produce concise, meaningful reports about changes in network conditions.

To track events inside their own network, operators collect measurements of data traffic, performance statistics, the internal topology, and equipment failures. The performance of a backbone network is especially vulnerable to *interdomain* routing changes that affect how data traffic travels to destinations in other Autonomous Systems (ASes). For example, a link failure in a remote AS could trigger a shift in how traffic travels through a network, perhaps causing congestion on one or more links. Fortunately, operators can gain additional visibility into the interdomain routing changes by monitoring the Border Gateway Protocol (BGP) decisions of routers at the periphery of their AS. In this paper, we address the challenge of analyzing a large volume of BGP update messages from multiple routers in real time to produce a small number of meaningful alerts for the operators.

In addition to the large volume of data, producing useful reports is challenging because: (i) BGP update messages show the changes in AS-level paths without indicating why or where they originated, (ii) a single network event (such as a failure) can lead to multiple update messages during routing protocol convergence, (iii) a single network event may affect routing decisions at multiple border routers, and (iv) a single event may affect multiple destination prefixes. Having a small number of reports that highlight only *important* routing changes is crucial to avoid overwhelming the operators with too much information. The reports should focus on routing changes that disrupt reachability, generate a large number of update messages, affect a large volume of traffic, or are long-lived enough to warrant corrective action. These concerns drive the design of our system. We have evaluated our system on two months of data from a tier-1 ISP and discovered several important problems that were previously unknown. Our system analyzes millions of

BGP update messages per day to produce a few dozen actionable reports for the network operators.

Despite some high-level similarities, our approach differs markedly from recent work on root-cause analysis of BGP routing changes [6, 8, 13, 15, 30]. These studies analyze streams of BGP update messages from vantage points throughout the Internet, with the goal of inferring the location and cause of routing changes. Instead, we consider BGP routing changes seen *inside* a single AS to identify—and quantify—the *effects* on that network. Realizing that root-cause analysis of routing changes is intrinsically difficult [27], we search only for explanations of events that occur close to the AS—such as internal routing changes and the failure of BGP sessions with neighboring domains—and mainly focus on alerting operators to the performance problems they can address. Hence, our approach is complementary to previous work on root-cause analysis, while producing results of direct and immediate use to network operators.

In the next section, we present background material on BGP, followed by an overview of our system in Section 3. In Section 4, we group BGP update messages into routing *events*. We identify persistently flapping prefixes and pinpoint the causes. In Section 5, we introduce the concept of a *route vector* that captures the best BGP route for each prefix at each border router. We identify five types of routing changes that vary in their impact on the traffic flow. In Section 6 we group events by type to identify frequently flapping prefixes, BGP session resets, and internal routing disruptions; we validate our results using RouteViews data, syslog reports, and intradomain topology data. In Section 7, we use prefix-level traffic measurements to estimate the impact of the routing changes. Section 8 shows that our system operates quickly enough to generate reports in real time. Section 9 presents related work, and Section 10 concludes the paper.

2 BGP Overview

The Border Gateway Protocol (BGP) [21] is the routing protocol that ASes use to exchange information about how to reach destination address blocks (or *prefixes*). Three key aspects of BGP are important for our study:

Path-vector protocol: Each BGP advertisement includes the list of ASes along the path, along with other attributes such as the next-hop IP address. By representing the path at the AS level, BGP hides the details of the topology and routing inside each network.

Incremental protocol: A router sends an advertisement of a new route for a prefix or a withdrawal when the route is no longer available. Every BGP update message is indicative of a routing change, such as the old route disappearing or the new route becoming available.

1. Ignore if the next hop is unreachable;
2. Highest local preference;
3. Shortest AS path;
4. Lowest origin type;
5. Lowest Multiple-Exit-Discriminator (MED) value among routes from same AS;
6. eBGP routes over iBGP routes;
7. Lowest IGP cost ("hot-potato");
8. Lowest router ID;

Table 1: BGP decision process

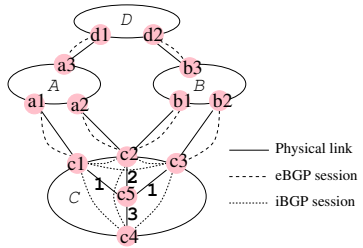


Figure 1: Interaction of routing protocols in AS C

Policy-oriented protocol: Routers can apply complex policies to influence the selection of the best route for each prefix and to decide whether to propagate this route to neighbors. Knowing why a routing change occurs requires understanding how policy affected the decision.

To select a single best route for each prefix, a router applies the decision process [21] in Table 1 to compare the routes learned from BGP neighbors. In backbone networks, the selection of BGP routes depends on the interaction between three routing protocols:

External BGP (eBGP): The border routers at the periphery of the network learn how to reach external destinations through eBGP sessions with routers in other ASes. A large network often has multiple eBGP sessions with another AS at different routers. This is a common requirement for two ASes to have a peering relationship, and even some customers connect in multiple locations for enhanced reliability. For example, Figure 1 shows AS C has two eBGP sessions with AS A and two eBGP sessions with AS B. As a result, there are three egress points to destinations in AS D.

Internal BGP (iBGP): After applying local policies to the eBGP-learned routes, a border router selects a single best route and uses iBGP to advertise the route to the rest of the AS. In the simplest case, each router has an iBGP session with every other router (*i.e.,* a *full-mesh* iBGP configuration). In Figure 1, the router $c4$ learns a two-hop AS path to destinations in AS D from three routers $c1$, $c2$, and $c3$.

Interior Gateway Protocol (IGP): The routers inside the AS run IGP to learn how to reach each other. The two most common IGPs are OSPF and IS-IS, which compute shortest paths based on configurable link weights. The

routers use the IGP path costs in the seventh step in Table 1 to select the *closest* egress point. In Figure 1, the number near each link inside AS C indicates the IGP cost of the link. Based on the decision rules, $c4$ prefers the routes through $c1$ and $c3$ over the route through $c2$ due to the smaller IGP path costs.[1]

The decision process in Table 1 allows us to compare two routes based on their attributes. We exploit this observation to determine whether a router switched from a better route to a worse route, or vice versa.

3 System Architecture

In this section, we describe how to track the BGP routing changes in an AS. Then, we present an overview of our system and describe the data we collected from a Tier-1 ISP backbone to demonstrate the utility of our tool.

3.1 Measurement Infrastructure

The routers at the edge of an AS learn BGP routes via eBGP sessions with neighboring domains, and then send update messages to other routers inside the AS via iBGP sessions. These border routers have complete visibility into external and internal routing changes. Ideally, each border router would provide a complete, up-to-date view of *all* routes learned from eBGP and iBGP neighbors. This data would allow our system to emulate the BGP decision process of each router, to understand why a router switched from one BGP route to another. Unfortunately, acquiring a timely feed of all eBGP updates received from neighboring ASes is difficult in practice.[2]

In this study, we analyze routing changes using only the data readily available in today's networks—a feed of the *best* route for each prefix from each border router. Our monitor has an iBGP session with each border router to track changes to the best route over time. A daily snapshot of the routing table from each border router is also collected to learn the initial best route for each prefix.

Since routing changes can have a significant effect on the distribution of the traffic over the network, traffic measurements are very useful for quantifying the *impact* of a routing change. In our measurement infrastructure, the monitor receives a feed of prefix-level traffic statistics from each border router. Because our analysis focuses on how routing changes affect the way traffic *leaves* the network, we collect the outgoing traffic on the edge links emanating from the border routers.

3.2 System Components

Our troubleshooting system analyzes BGP routing changes visible from inside a single AS and quantifies the effects on the network. The system is designed to operate *online* so operators may take corrective actions to improve network performance. For ease of presentation, we describe the functionality of our system in four distinct stages, as illustrated in Figure 2:

RouteTracker (Section 4): The first module merges the streams of BGP updates from the border routers and identifies routing *events*—groups of update messages for the same prefix that occur close in time. Along the way, the module identifies prefixes that flap continuously.

EventClassifier (Section 5): The second module classifies the routing events in terms of the kind of routing change and the resulting impact on the flow of traffic through the network. For example, we define a category called *internal disruption* that pinpoints the events caused by internal topology changes.

EventCorrelator (Section 6): The third module identifies related events by clustering over time and prefixes. In contrast to previous studies [6, 8, 13, 15, 30], we focus mainly on events that occur very close to the network (*e.g.,* eBGP session resets or internal disruptions) and have a significant impact on traffic. In addition, our correlation algorithms consider whether the border routers switched from a better route to a worse route, or vice versa—information not readily available in eBGP data feeds used in previous work on BGP root-cause analysis.

TrafficMeter (Section 7): The last module estimates the impact of routing changes on the flow of traffic, to draw the operators' attention to the most significant traffic shifts. Using prefix-level measurements of the traffic leaving the network, TrafficMeter computes a traffic weight that estimates the relative popularity of each prefix. The module predicts the severity of each event cluster by adding the weights of the affected prefixes.

In moving from raw updates to concise reports, we apply time windows to combine related updates and events, and thresholds to flag clusters with significant traffic volumes. We use our measurement data and an understanding of BGP dynamics to identify appropriate time windows; the threshold values reflect a trade-off between the number and significance of the disruptions we report.

3.3 Applying the System in a Tier-1 ISP

We have applied our prototype to a Tier-1 ISP backbone with hundreds of border routers connecting to customer and peer networks. Although we would ideally have iBGP sessions with all border routers, we could only collect data from the routers connecting to peer networks. Still, the BGP routing changes at these routers give us a unique view into the effects of BGP routing changes in the larger Internet on the ISP network. In addition, these border routers receive reachability information about customer prefixes via iBGP sessions with other routers, allowing us to analyze changes in how

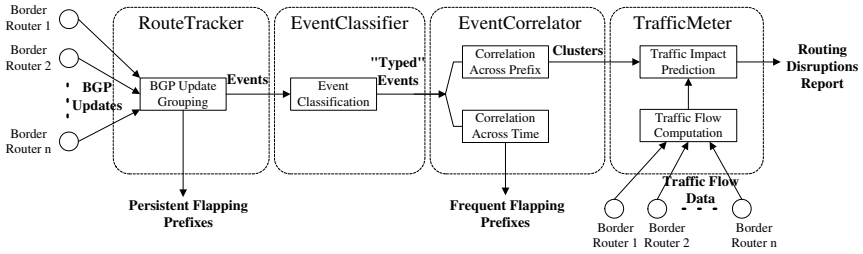


Figure 2: System design

Component	Reduction	Factor
RouteTracker	updates → events	15.2
EventCorrelator	events → clusters	31.7
TrafficMeter	clusters → "important" clusters	327.6
Total	updates → "important" clusters	158460

Table 2: Incremental information reduction

these border routers would direct traffic via customers. On a few occasions, our monitor experienced a temporary disruption in its iBGP connectivity to a border router; we preprocessed the BGP feeds as suggested in [22, 29] to remove the effects of these session resets.

The traffic data is collected from every border router by enabling Cisco's Sampled Netflow [1] feature on all links. To reduce the processing overhead, flow records are sampled using techniques in [19]. Although sampling introduces inaccuracies in measuring small traffic volumes, this does not affect our system since we only use the traffic data to identify large traffic disruptions.

As shown in Table 2, our system significantly reduces the volume of data and produces only a few dozen large routing disruptions from millions of BGP updates per day from the periphery of the network. "Important" clusters in the table are clusters that affect more than 1% of total traffic volume in the network. In the remainder of the paper, we present detailed results from the routing and traffic data collected continuously from August 16, 2004 to October 10, 2004—an eight-week period.

4 Tracking Routing Changes

In this section, we describe how we transform raw BGP update messages into routing events. We merge streams of updates from many border routers and identify changes from one stable route to another by grouping update messages that occur close in time. Along the way, we generate a report of prefixes that flap continuously.

4.1 Grouping BGP Updates into Events

A single network disruption, such as a link failure or policy change, can trigger multiple BGP messages as part of

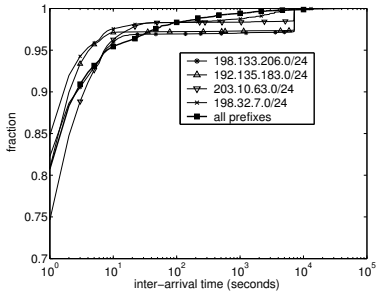


Figure 3: CDF of the BGP update inter-arrival time

the convergence process [3, 4]. The intermediate routes are short-lived and somewhat arbitrary, since they depend on subtle timing details that drive how the routers explore alternate paths. To generate reports for the operators, we are interested in the change from one stable route to another rather than the details of the transition. As such, we group BGP updates for the same prefix that occur close together in time. Although previous studies, in particular BGP root-cause analysis, have followed a similar approach [6, 8, 13, 20, 22], we group the updates across *all* of the border routers since a single network disruption may cause multiple border routers to switch to new routes, and we wish to treat these as a single event.

We define an *event* as a sequence of BGP updates for the same prefix from any border router where the inter-arrival time is less than a predefined *event timeout*. Careful selection of the event-timeout value is important to avoid mistakenly combining unrelated routing changes or splitting a single change into two events. An appropriate event-timeout value can be determined by characterizing the inter-arrival time of BGP updates in the network. For a controlled experiment, we analyze the inter-arrival times of BGP updates for public *beacon* prefixes that are advertised and withdrawn every two hours [17]; we also study the dynamics of the entire set of prefixes.

Figure 3 presents the cumulative distribution of the inter-arrival time of BGP updates for four beacons received from all of the border routers during a three-week period starting August 16, 2004, with the x-axis plot-

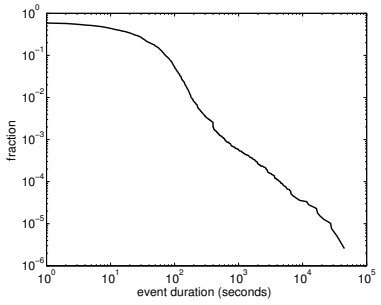


Figure 4: CCDF of event duration on a log/log scale

ted on a logarithmic scale. More than 95% of the inter-arrival times are within a few tens of seconds; then the curves flatten until the inter-arrival time is around 7,000 seconds reflecting the two-hour advertisement period. In addition, previous studies have shown that the path-exploration process is often regulated by a 30-second *MinRouteAdvertisementInterval* (MRAI) timer [14]. As such, we choose an event timeout of 70 seconds, allowing the difference between the arrival times of updates at different vantage points to be as large as two MRAIs plus a small amount of variance. Looking across all prefixes in our dataset, about 98% of the updates arrive less than 70 seconds after the previous update.

4.2 Detecting Persistent Flapping

Certain prefixes never converge to a stable path due to persistent routing instabilities. Persistent flapping disrupts the reachability of the destination and imposes a significant BGP processing load on the routers, making it important for operators to detect and fix these problems. However, if we group updates for a flapping prefix using a 70-second timeout, the grouping process would continue indefinitely. Instead, we generate a report once a sequence of updates exceeds a *maximum* duration, defined as the *convergence timeout*.

The convergence-timeout value should be large enough to account for reasonable convergence delays and yet small enough to report persistent flapping to the operators in a timely fashion. To identify an appropriate value, Figure 4 plots the complementary cumulative distribution function (CCDF) of event duration for the BGP updates in our network, with both axes on a logarithmic scale. More than 99% of events last less than a few hundred seconds, consistent with the findings in [3] that BGP typically takes less than three minutes to converge. As such, we select a convergence-timeout value of 600 seconds (10 minutes) for reporting flapping prefixes.

By applying our RouteTracker module to eight weeks of measurement data, we generated reports for about 23 prefixes per day, on average, though the number was as

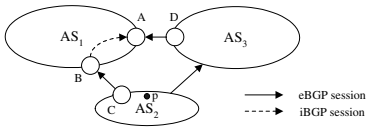


Figure 5: Persistent flapping due to failure of link B–C

low as 7 on one day and as high as 46 on others. These persistently flapping prefixes were responsible for 15.2% of the total number of BGP update messages over the two-month period, though the proportion varied significantly from day to day (from 3.2% to 44.7%). These results were especially surprising given that all of the border routers were running route-flap damping [5], which is meant to suppress repeated updates of the same prefix. We identified three main causes of persistent flapping:

Unstable interface/session: Using syslog data [16] from the border routers, we determined that 3% of these updates (0.456% of the total number of updates) were caused by repeated failures of a flaky edge link or eBGP session. The prefixes were advertised each time the link/session came online, and withdrawn when the link/session failed. In Figure 5, the routers in AS_1 prefer the BGP route advertised by the customer AS_2 over the BGP route advertised by the peer AS_3. However, a flaky link between routers B and C would lead the routers in AS_1 to repeatedly switch between the stable route via AS_3 and the unstable route via AS_2. Route-flap damping did not stop AS_1 from using the unstable route from AS_2 for two reasons: (i) today's routers reinitialize the damping statistics associated with an eBGP session after a session reset and (ii) routers do not perform route-flap damping on iBGP sessions. In the short term, operators could respond to these cases by disabling (and ultimately repairing) the flaky link or session; in the longer term, router vendors could change the implementation of route-flap damping to prevent the persistent flapping.

MED oscillation: Through closer inspection of the BGP update messages and discussions with the operators, we determined that 18.3% of these updates (2.78% of the total) were caused by protocol oscillation due to the Multiple Exit Discriminator attribute. Unlike the other steps in the decision process in Table 1, the MED comparison is applied only to routes with the same next-hop AS. As a result, the BGP decision process does *not* impose an ordering on the routes in the system: a router may prefer route a over route b, b over c, and c over a. In the absence of an ordering of the routes, the routers may switch continuously between routes [18, 25]. Upon detecting a MED oscillation problem, the operators can request that the neighboring AS use a different mechanism to express its preferences for where it wants to receive the traffic destined for these prefixes (*e.g.,* RFC 1998 [10]).

Conservative flap-damping parameters: The remaining 78.6% of these updates (11.9% of the total) correspond to repeated advertisements and withdrawals by a neighboring AS. By inspecting the configuration of the routers, we verified that the flap-damping parameters assigned for these prefixes were not sufficient to dampen the instability. Using different parameters for different prefixes is not uncommon and is, in fact, recommended [2]. For example, ASes are advised to more heavily penalize the (many) smaller address blocks and to disable damping on critical prefixes (*e.g.,* the subnets that contain the Internet's root DNS servers). Upon noticing persistent flapping that is evading the damping algorithm, the operator could contact the neighboring AS to investigate the root cause or tune the router configuration to apply more aggressive damping parameters.

5 Classifying Routing Changes

In this section, we describe how we classify events to generate useful reports for the operators and to facilitate the clustering of related events in the next section. Since the current measurement infrastructure collects the BGP data only from the border routers connecting to peer networks, the following analysis is applied to the prefixes learned exclusively from peer ASes.

5.1 Merging Routes from Border Routers

To handle the large volume of BGP data arriving from the many border routers, EventClassifier needs a succinct representation of the routing state as it evolves over time. Rather than considering every BGP attribute, we focus our attention on how traffic entering at a border router would leave the AS en route to the destination prefix p. A border router BR_j may select a route R_p^j learned directly from one of its eBGP neighbors; in this case, we say that BR_j has route R_p^j with the next-hop address $nhop_p^j$ corresponding to the eBGP neighbor and a $flag_p^j$ of **e** for external. Alternatively, a border router BR_j may select as R_p^j a route learned via iBGP from another border router, resulting in a next-hop address $nhop_p^j$ of the remote border router and a $flag_p^j$ of **i** for internal. In a network with n border routers $BR_1, BR_2, \ldots, BR_n$, we have a route vector (r-vector) for prefix p of

$$RV_p = \langle R_p^1, R_p^2, ..., R_p^n \rangle$$

where the jth element $R_p^j = (nhop_p^j, flag_p^j)$ represents the *best* route for prefix p at router BR_j. By analyzing the evolution of RV_p, we can identify and classify the routing changes that affect how traffic leaves the AS, while ignoring changes in other BGP attributes (*e.g.,* downstream AS path or BGP community) that are beyond the operators' control.

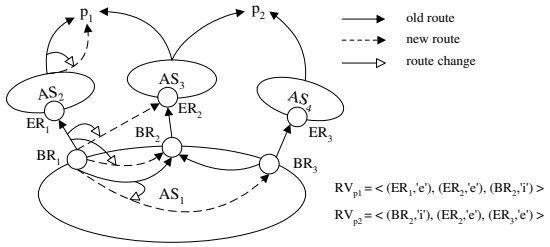


Figure 6: R-vector element changes

5.2 Classifying Routing Events

When the network changes from one set of stable routes to another, comparing the old and new r-vectors (RV_p^{old} and RV_p^{new}, respectively) sheds light on the reason for the change and the effects on the traffic. We first describe the types of changes that each border router might experience and then present five event categories that consider the behavior across all of the routers.

5.2.1 Types of Events at One Border Router

To illustrate the types of routing events, Figure 6 shows examples for two destination prefixes. For prefix p_1, border routers BR_1 and BR_2 have eBGP-learned routes through AS_2 and AS_3, respectively; border router BR_3 selects an iBGP-learned route through BR_2. For prefix p_2, border routers BR_2 and BR_3 have eBGP-learned routes through AS_3 and AS_4, respectively; border router BR_1 selects an iBGP-learned route through BR_2. The dashed lines represent different ways an event can affect BR_1's routing decision, as summarized in Table 3:

No change: The border router BR_j may undergo a transient routing change only to return to the same stable best route. More generally, the BGP route may change in some attribute that is not captured in R_p^j. In Figure 6, a change in how AS_2 reaches p_1 does not necessarily change BR_1's decision to direct traffic via AS_2. For all of these scenarios, traffic entering the network at router j destined for the prefix p would continue to flow through the AS in the same way.

Internal path change: An internal event may cause a router to switch from one egress point to another. In this case, router j uses an iBGP-learned route before and after the routing change (*i.e.,* $flag_p^{j,new} = flag_p^{j,old} =$**i**) but with a different next-hop router (*i.e.,* $nhop_p^{j,new} \neq nhop^{j,old}$). In Figure 6, a change in the IGP topology could make BR_1 see BR_3 as the *closest* egress point for reaching prefix p_2, instead of BR_2.

Loss of egress point: An external event may cause a route to disappear, or be replaced with a less attractive alternative, forcing a border router to select an iBGP route. In this case, a router BR_j has $flag_p^{j,old} =$**e** and

Type of Change for R_p^j	Definition
No change	$flag_p^{j,old} = flag_p^{j,new}$ $nhop_p^{j,old} = nhop_p^{j,new}$
Internal path change	$flag_p^{j,old} = flag_p^{j,new} =$**i**, $nhop_p^{j,old} \neq nhop_p^{j,new}$
Loss of egress point	$flag_p^{j,old} =$**e**, $flag_p^{j,new} =$**i**
Gain of egress point	$flag_p^{j,old} =$**i**, $flag_p^{j,new} =$**e**
External path change	$flag_p^{j,old} = flag_p^{j,new} =$**e**, $nhop_p^{j,old} \neq nhop_p^{j,new}$

Table 3: The types of change for r-vector element R_p^j

Event Category	Events	Updates	Upd./Ev.
Distant/transient disruption	50.3%	48.6%	12.6
Internal disruption	15.6%	3.4%	2.9
Single external disruption	20.7%	7.9%	5.0
Multiple external disruption	7.4%	18.2%	32.0
Loss/gain of reachability	6.0%	21.9%	47.9

Table 4: Event distribution in updates

$flag_p^{j,new} =$**i**. In Figure 6, suppose AS_2 withdraws its route for p_1 and that BR_1 has no other eBGP-learned routes; then, BR_1 would select the iBGP-learned route from BR_2. This routing change would force the traffic that used to leave the network at BR_1 to shift to BR_2.

Gain of egress point: An external event may cause an eBGP-learned route to appear, or be replaced with an attractive alternative, leading a border router to switch from an iBGP-learned route to an eBGP-learned one. In this case, a router BR_j has $flag_p^{i,old} =$**i** and $flag_p^{j,new} =$**e**. In Figure 6, suppose AS_2 starts advertising a route to p_1 again; then, BR_1 would start using the eBGP-learned route, causing a shift back to BR_1.

External path change: An external event may cause a router to switch between eBGP-learned routes with different next-hop ASes. In this case, the $flag_p^j$ remains at **e** while the next hop changes (*i.e.,* $nhop_p^{j,new} \neq nhop_p^{j,old}$). In Figure 6, suppose AS_2 withdraws the route for p_1, causing BR_1 to switch to an eBGP-learned route from AS_3. Then, BR_1 would start directing traffic to a different egress link at the same router.

5.2.2 Classes of Route-Vector Changes

Since each of the n elements in the r-vector can have five different types of changes, routing events could fall into 5^n different categories, which would be extremely unwieldy for generating reports for network operators. Instead, we classify the events based on the *severity* of the impact on the traffic, leading to five disjoint categories:

Distant/transient disruption: Some events do not have any influence on the flow of traffic through the AS. We define an event as *distant or transient disruption if each element of the r-vector has "no change."* A distant routing change that occurs more than one AS hop away does not affect the R_p^j values. A transient disruption may cause temporary routing changes before the border routers converge back to the original BGP routes. These events are worthwhile to report because the downstream routing change may affect the end-to-end performance (*e.g.,* by changing the round-trip time for TCP connections) and the convergence process may lead to transient performance problems that can be traced to the routing event. As shown in Table 4, this category explains about half of the events and half of the BGP update messages; these events trigger an average of 12 or 13 update messages for the BGP convergence process.

Internal disruption: An internal event can cause a router to switch from one internally-learned route to another. We define an event as an *internal disruption if the change of each of the elements in its r-vector is either of type "no change" or of type "internal path change", with at least one element undergoing an "internal path change."* Caused by a change in the IGP topology or an iBGP session failure, these events are important because they may cause a large shift in traffic as routers switch from one egress point to another [27, 28]. As shown in Table 4, internal disruptions account for about 15% of the events and just 3.4% of the updates; on average, an internal event triggers just a few iBGP update messages as some routers switch from one existing route to another.

Single external disruption: Some events affect the routing decision at a single border router for an eBGP-learned route. We define an event as a *single external disruption if only one r-vector element has a change of type "loss of egress point," "gain of egress point," or "external path change."* Typically, an ISP has eBGP sessions with a neighboring AS at multiple geographic locations, making it interesting to highlight routing changes that affect just one of these peering points. These kinds of events cause a shift in traffic because routers are forced to select an egress point that is further away [12]. For example, a single external disruption may arise because an eBGP session between the two ASes fails, forcing the border router to switch to a less-attractive route. As shown in Table 4, these disruptions account for over 20% of the events and nearly 8% of the updates; since these localized events affect a single router, the number of update messages per event is limited.

Multiple external disruptions: In contrast to the previous category, some events affect more than one border router. We define an event as a *multiple external disruption if multiple r-vector elements have a change of type "loss of egress point," "gain of egress point," or "external path change," and the r-vector includes at least one eBGP-learned route before and after the event.*[3] In Figure 6, if the owners of prefix p_1 changed providers to start using AS_4 instead of AS_2 and AS_3, every border routers

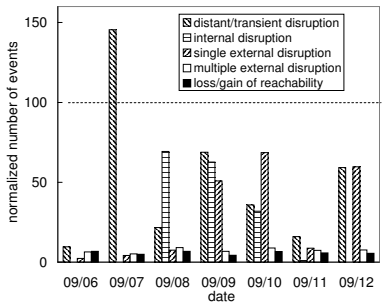


Figure 7: The (normalized) # of daily events by category.

in AS_1 would experience a disruption. As shown in Table 4, this category accounts for just over 7% of events and 18% of updates; the large number of update messages stems from the convergence process where multiple border routers must explore alternate routes.

Loss/gain of reachability: An event may cause a prefix to disappear, or become newly available. We define an event as *loss of reachability if every r-vector element with an external route experiences a "loss of egress point."* A loss of reachability is extremely important because it may signify a complete loss of connectivity to the destination addresses, especially if the routers have no route for other prefixes (*e.g.,* supernets) covering the addresses. Similarly, we define an event as *gain of reachability if initially no eBGP-learned routes exist and at least one r-vector element experiences a "gain of egress point."* In some cases, the *gain* of reachability is indicative of a problem, if the network does not normally have routes for that prefix. For example, a neighboring AS may mistakenly start advertising a large number of small subnets; overloading the memory resources on the router may have dire consequences, such as crashing the network [7]. As shown in Table 4, this category accounts for 6% of the events and nearly 22% of the update messages; the gain or loss of reachability often triggers a large number of update messages as every border router explores the many alternate routes.

Overall, the severity of the external events increases from single external disruptions to multiple external disruption, and ultimately to loss/gain of reachability. In general, the number of events in the "loss/gain of reachability" and "multiple external disruption" is stable over time, whereas the other categories vary significantly. Figure 7 shows the number of daily events (where 100 represents the average number of events per day over the eight-week study) for each event category during the week of September 6-12, 2004. For example, September 7 had a large number of distant/transient disruptions, and some days see a much larger number of internal disruptions and single external disruptions than others. The high variability arises from the fact that network disruptions can occur at arbitrary times and may affect a large number of destination prefixes, as discussed in the next section. Given the high variability in the number and type of events, predicting them in advance and overprovisioning for them is very difficult, making it even more important for operators to learn about disruptions as they occur to adapt the configuration of the network.

6 Grouping Related Events

In this section, we describe how to identify related events across *time* and *prefixes*. By clustering events across time for the same prefix, we identify destination prefixes that have unstable routes. By clustering events of the same type across prefixes, we group events that appear to have a common cause. We present techniques to identify groups of prefixes affected by hot-potato routing changes and eBGP session resets, which are responsible for many of the large clusters. We validate our inferences using RouteViews data [24], syslog reports [16], and an independent analysis [28] of internal topology changes.

6.1 Frequently Flapping Prefixes

Some destination prefixes undergo frequent routing changes that introduce a large number of events in a relatively short period of time. In contrast to the persistent flapping analyzed in Section 4.2, these routing changes occur at a low enough rate to span multiple events. For example, a prefix may have a long-term instability due to flaky equipment that fails every few minutes, falling outside of our 70-second window for grouping BGP updates into events. Even if the equipment fails at a higher rate, the BGP updates may be suppressed periodically due to route-flap damping [5], leading to multiple events. Identifying these slowly *frequently flapping* prefixes is important for addressing long-term reachability problems and for reducing the number of BGP updates the routers need to handle.

To identify frequently flapping prefixes, we group events for the same destination prefix that occur *close together in time* (with an inter-arrival time less than $thresh_T$), and flag cases where the *number of events exceeds a predefined threshold* (max_count). We implement this heuristic by keeping track of each prefix that has had an event in the last $thresh_T$ seconds, along with the time of the last event and a count of the total number of events. Upon learning about a new event from RouteTracker, we check if the prefix has experienced an event in the last $thresh_T$ seconds and update the timestamp and counter values; once the counter exceeds max_count, we generate a report.

Since route changes can happen on virtually any timescale, the parameters $thresh_T$ and max_count

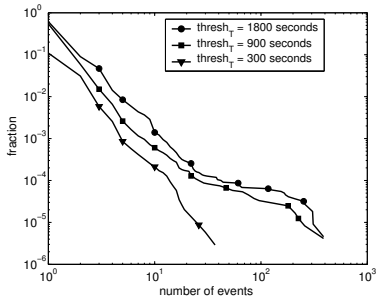


Figure 8: CCDF of the number of events per cluster for event correlation across time

should be set to highlight the most unstable prefixes without generating an excessive number of reports. Figure 8 shows the complementary cumulative distribution of the number of events per cluster over our eight-week measurement period. For all three values of $thresh_T$, more than 99% of the clusters have fewer than ten events; still, a small number of very large clusters exist. Having a very small $thresh_T$ might cause our system to overlook some unstable prefixes with a long cycle between routing changes. For example, a prefix that has a routing change every ten minutes would not be detected by a $thresh_T$ of 300 seconds. Based on the results in Figure 8, we assign $thresh_T$ to 900 seconds and max_count to 10 to draw attention to the small number of very unstable prefixes.

In our analysis, the percentage of events caused by frequently flapping prefixes varies from day to day from a low of 0.41% to a high of 32.78%, with an average of 3.38%. Most of these events are in category "loss/gain of reachability." We believe that frequent flapping tends to originate near the destination, making these instabilities visible to other ASes. To validate our inferences, we applied our heuristic for identifying frequently flapping prefixes to the BGP data from RouteViews [24]. For the week of September 26 to October 2, 2004, all 35 prefixes we identified were also flapping frequently in at least one other vantage point in the RouteViews data. Whether (and how) operators react to frequently flapping prefixes depends on the network responsible for the problem. If the frequent flapping comes from one of the ISP's own customers, the operators may be able to work with the customer to identify and fix the problem. If the flapping comes directly from a peer network (or one of the peer's customers), the operators may contact the peer to request that the peer address the problem.

6.2 Disruptions Affecting Multiple Prefixes

A single disruption (such as a link failure or a policy change) may affect multiple prefixes in a similar way, in a very short period of time. Grouping these prefixes together magnifies the visibility of the common effects and substantially reduces the number of reports for the operators. The five categories identified in Section 5.2 provide an effective way to identify prefixes affected in a "similar way." In addition, we also consider whether the border routers changed from a better route to a worse route, a worse route to a better route, or between two equally-good routes, in terms of the first six steps of the decision process in Table 1. This distinction gives us insight into whether the old route was withdrawn (or replaced by a less-attractive route), the new route recently appeared (or was replaced by a more-attractive route), or the router switched between two comparable routes (*e.g.,* because of a change in the IGP path costs).

In particular, we group events for different destination prefixes that (i) belong to the same category (using the taxonomy from Section 5.2), (ii) undergo the same kind of transition (from better to worse, or worse to better), and (iii) start no more than $thresh_P$ seconds after the first event. We consider the start time of the events because the first update is most likely to be directly triggered by the network event. We implement this heuristic by keeping track of the identifying information for each cluster (*i.e.,* the event category and the kind of transition) as well as the time of the first event and a count of the number of events. Upon generating a new event, we check if the event matches with the identifying information and arrives within $thresh_P$ seconds after the first event in the cluster. The correlation process adopts a clustering algorithm similar to those used in previous BGP root-cause analysis studies [6, 8, 13].

Setting $thresh_P$ too small runs the risk of splitting related events into two clusters. If a network disruption affects a large number of prefixes, the effects could easily spread over several tens of seconds. For example, a BGP session failure or hot-potato routing disruption that affects tens of thousands of prefixes requires the router to send numerous update messages, which could easily take up to a minute [28]. To account for these effects, we carefully select a value of 60 seconds for $thresh_P$ after a study of the duration traditional routing changes (*e.g.,* session resets) normally take to affect all of their related prefixes. Since $thresh_P$ is used to compare the start times of the two events, our heuristic cannot assume that a cluster is complete once the current time (the time of newly arrived BGP update in the system) is $thresh_P$ after the time of the first event in the cluster since an event may still be "in progress." Knowing that an event lasts at most the convergence timeout (from Section 4.2), in our heuristic, each cluster waits for a total of $thresh_P + convergence\_timeout$ to ensure that no ongoing, correlated events should be included in the cluster. In total, then, our heuristic waits for 660 seconds before declaring a cluster complete.[4]

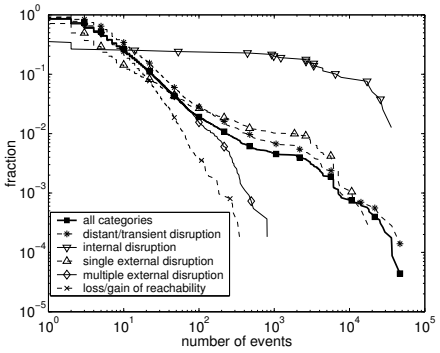


Figure 9: CCDF of the number of event per cluster for event correlation across prefixes

Figure 9 shows the effectiveness of clustering in combining related events. The graph plots the complementary cumulative distribution of the number of events per cluster over the eight-week period, on a log-log scale. Although 99% of the clusters have less than a hundred events (as shown in the "all categories" curve), a few clusters have a tremendous number of events. Meanwhile, the curves for different categories of events have distinctive characteristics. The categories "multiple external instability" and "loss/gain of reachability" have much smaller clusters, while the other three categories have some very large clusters with tens of thousands of affected prefixes. The categories "internal disruption" and "single external disruption" tend to have larger clusters than the other categories. Next, we show that these very large clusters stem from hot-potato routing changes and eBGP session resets, respectively.

6.2.1 Hot-Potato Changes

According to the BGP decision process in Table 1, a router selects among multiple equally good BGP routes (*i.e.,* routes that have the same local preference, AS path length, origin type, MED value, and eBGP vs. iBGP learned) the one with the smallest IGP cost. Such routing practice is called *hot-potato* routing [28]. An IGP topology change can trigger routers in a network to select a different *equally good* BGP route for the same prefix, and these changes may affect multiple prefixes. This section describes the routing disruptions caused by these hot-potato changes.

"Hot-potato" changes only affects the egress points each router selects for the prefixes. As the event classification in Section 5.2, it results in "internal disruptions" to the network. After the correlation process, the event cluster in category "internal disruption" magnifies the impact of the "hot-potato" changes. When these kinds of disruptions occur, the operators need to know which routers and prefixes are affected to gauge the significance of the event. Such information can be obtained by comparing the old and new r-vectors for all of the events in the cluster because each element in the r-vector carries the next-hop address for the corresponding router.

A previous study [28] proposed a heuristic for identifying hot-potato routing changes at a single router, based on a single stream of BGP updates from that router and data from an IGP topology monitor. Applying this technique to specific ingress routers allowed us to make direct comparisons between the two approaches. For the period from August 16 to September 30, 2004, over 95% of the large clusters (*i.e.,* clusters with more than 1000 events) of internal disruptions identified by our system are also identified using the technique in [28]. Inspecting the other 5% of cases in more detail, we discovered that these clusters corresponded to the restoration of a link in the network, where the failure had caused a previous hot-potato routing change that was detected using both techniques. As such, we believe that these disruptions are hot-potato routing changes that were not detected by the heuristic in [28].

6.2.2 eBGP Session Resets

The failure or recovery of an eBGP session can cause multiple events that affect the eBGP-learned routes from one neighbor at a single border router. Upon losing eBGP connectivity to a neighbor, a border router must stop using the routes previously learned from that neighbor and switch to less-attractive routes. The border router may switch to an eBGP-learned route from a different neighbor, if such a route exists; this would result in an "external path change" for the destination prefix. Alternatively, the router may have to switch to an iBGP-learned route from a different border router; this would result in a "loss of egress point" for the destination prefix. When the session recovers, the border router learns the BGP routes from the neighbor and switches back to the eBGP-learned routes advertised by this neighbor for one or more destination prefixes (causing either an "external path change" or a "gain of egress point").

To identify a session failure, we first group events that (i) belong to the category "single external disruption," (ii) have an *old* route with the same border router and neighbor (*i.e.,* the same $R_p^{j,old}$), (iii) have a routing change that goes from better to worse, and (iv) occur close together in time. However, this is not enough to ensure that the session failed, unless the router has stopped using most (if not all) of the routes previously learned from that neighbor. As such, we also check that the number of prefixes using the neighbor has decreased dramatically.[5] Similarly, to identify a session recovery, we first group events that (i) belong to the category "single external disruption," (ii) have a *new* route with the same border

router and neighbor (*i.e.,* the same $R_p^{j,old}$), (iii) have a routing change that goes from worse to better, and (iv) occur close together in time, and also involve a significant increase in the number of prefixes associated with that neighbor, back to the expected level.

Applying our heuristic to the "single external disruption" clusters that contain more than 1000 events, we found that 95.7% of these large clusters were linked to an eBGP session going up or down. To validate our inferences, we consulted the syslog data [16], which reports when the status of a BGP session changes. The syslog data confirmed more than 95% of our inferences. Our inferences not only captured all of the resets in syslog but identified a few disruptions that were not reported by syslog. Interestingly, we sometimes found that our analysis suggests that the session failure occurred up to ten seconds *before* the entry in the syslog data. After checking for possible timing discrepancies between the BGP and syslog data, we speculate that the remote AS is shutting down the BGP session in a graceful manner by first *withdrawing* all of the routes before actually disabling the session. This practice highlights the importance of using an algorithm such as ours even when syslog data are available.[6] A complete loss of the routes from a neighbor does *not* necessarily arise only from a session failure. Instead, the neighbor's router may be reconfigured with a new policy (*e.g.,* that withdraws the previous routes) or lose connectivity to other routers in its own network. These kinds of disruptions could have a significant impact on traffic inside an AS, and would not generate a syslog report. The influence of large disruptions on the traffic is explored in more detail in the next section.

7 Estimating Traffic Impact

We now describe the final component of the system—TrafficMeter which allows us to estimate the traffic impact of the routing disruptions produced by the Event-Correlator. Although the traffic volume on a link typically varies gradually across days and weeks, sudden changes in traffic can lead to congestion in some parts of the network. A recent study [26] shows BGP routing disruptions are responsible for many of the largest traffic shifts in backbone networks. Below we first discuss how we compute traffic weights to estimate the impact on traffic and then focus on two types of routing disruptions with the most impact.

7.1 Computing Traffic Weights

TrafficMeter aggregates the Netflow data [1] collected on the outgoing links to compute prefix-level traffic statistics. For each destination prefix, we define a *traffic weight* that corresponds to the percentage of traffic destined to that prefix across the overall traffic volume in the network. In essence, the weight corresponds to the relative popularity of the prefix. Since the proportion of traffic destined to each prefix changes over time, we compute the weights over a sliding time window (*e.g.,* the last month). The weights allow us to estimate the potential impact of a cluster of routing events by considering the sum of the weights for all prefixes in the cluster. Although the weights do not capture the variations in traffic per prefix across time and location, they do provide a simple way to flag routing disruptions that affect clusters of prefixes that attract a high volume of traffic.

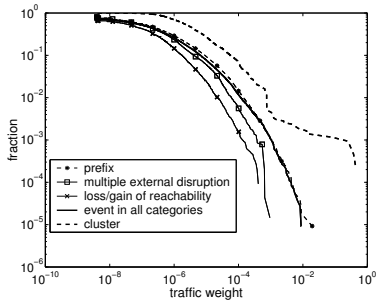


Figure 10: CCDF of traffic weight

In Figure 10, we plot the complementary cumulative distribution of traffic weight of a prefix, an event, and an event cluster over the eight-week period of our study. The "prefix" curve shows the significant differences in popularity of the prefixes, consistent with previous studies [11, 22]. Interestingly, the "event in all categories" curve looks largely the same, suggesting that routing events affect prefixes across the entire range of popularities. This occurs because the many events in categories "distant/transient disruption," "single external disruption," and "internal disruption" tend to affect a wide range of destination prefixes, largely independent of their popularity; the curves for these three categories of events are not shown, as they look almost identical to the "prefix" and "event in all categories" curves. In contrast, the curves for events in categories "multiple external disruption" and "loss/gain of reachability" suggest that these events tend to involve prefixes that receive less traffic.

The "cluster" curve plots the distribution of traffic weight across the event clusters. As expected, a cluster tends to have a large traffic weight since it combines one or more related events. The tail of the curve suggests that a small number of clusters are responsible for a significant portion of the large traffic shifts. Meanwhile, our results reveal that these "significant" clusters have a large number of events, implying the routing change affects many prefixes. Our system observes a few dozen such large clusters each day and highlights them for the

network operators for their attention. We use the threshold of 1% for traffic weight to signal significant routing disruptions, since the vast majority of clusters fall below that threshold. This avoids operators focusing their attention on the many BGP disruptions that affect a very small fraction of the traffic.

7.2 Disruptions With Large Weights

We now discuss our empirical findings using TrafficMeter based on our eight weeks of measurement data. Interestingly, most big events in terms of the amount of traffic weight are single external disruptions and internal disruptions. Thus, we focus on those in Figure 11 showing the duration of a routing disruption relative to the corresponding traffic weight of the affected prefixes for clusters with traffic weight larger than 1%. On average, internal disruptions (*e.g.,* hot potato changes) result in larger traffic weights than single external disruption (*e.g.,* session resets), because internal routing disruptions usually affect multiple locations. They also appear to have longer durations than single external disruptions. Long-lived events allow operators to adapt routing configurations as needed to alleviate possible network congestion. Our tool highlights only a few critical events which are both long-lived and expected to affect a large amount of traffic. This helps focus operators' attention on routing disruptions where mitigation actions, such as tuning the routing protocol configuration, might be necessary.

Figure 11 also shows that our tool captures some large disruptions that are short-lived, lasting 30 seconds to a few minutes. In addition to most of the "single external disruption" points in the graph, these short-lived disruptions include many large clusters in the "distant/transient disruption"; this category accounts for 78.8% of all event clusters with traffic weight higher than 1%. These clusters involve events that start and end with the same route vectors, with some sort of transient disruption in between. Although short-lived traffic shifts do not have a sustained impact on network load, users may encounter brief periods of degraded performance that could be traced to these disruptions. Interestingly, these short-lived traffic shifts are extremely difficult to detect using conventional measurement techniques, such as SNMP and Netflow, that aggregate traffic statistics on the timescale of minutes. In contrast, our troubleshooting system can identify short-lived routing disruptions that may have large effects on user performance.

8 System Evaluation

In this section, we demonstrate that our system imposes a small amount of memory and CPU processing overhead to run in real time on a commodity computing platform.

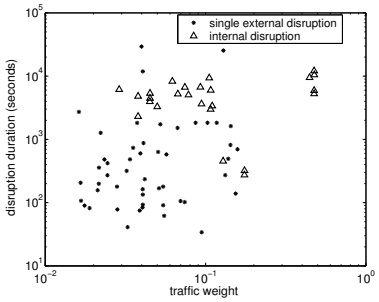


Figure 11: Routing disruption durations vs. traffic weights

Throughout the evaluation of our system on eight weeks of data, the system memory footprint never exceeded 900 Megabytes and every interval of 70 seconds of BGP updates was processed in less than 70 seconds.

We characterize the system performance through an off-line emulation over the past measurement data. Due to operational concerns, our system could not access the collected data in real-time. Instead, we stored the measurements locally and replayed the data in our tool. We ran our tool on a Sun Fire 15000 equipped with several 900 MHz Ultrasparc-II processors. Only one processor was used during the experiments. We evaluate the system using two metrics: *memory usage* and *execution speed*.

8.1 Memory Usage

The memory usage in our troubleshooting system consists of two parts: *static* usage and *dynamic* usage. The static memory is allocated to store the best route for each border router and destination prefix. In the core of today's Internet, each router learns reachability information for about 160,000 prefixes (also confirmed by RouteViews [24]). The total static memory usage in our system is about 600 Megabytes.

Dynamic memory, on the other hand, is allocated to maintain the data structures continuously created in response to the arrival of BGP updates. The essential data objects kept in the system are clusters, whose memory are dynamically allocated and reclaimed during the process as discussed in Section 6. In processing the eight weeks of measurement data, the dynamic memory footprint of the system never exceeded 300 Megabytes.

8.2 Execution Speed

We measure how quickly the system processes the BGP updates. Because the progression of each BGP update in the system varies depending on the expiration condition of several timers, we have conducted the experiment for each BGP update sequence within a fixed time interval called *epoch*, rather than characterizing the execution la-

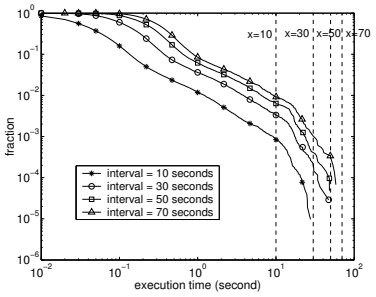


Figure 12: System execution speed

tency of each individual BGP update. During each test, we randomly selected a starting point in the eight-week BGP update sequence and then divided the subsequent BGP update stream into non-overlapping epochs. Then we measured the execution time for each epoch of a fixed epoch interval. We varied the epoch interval among the values of 10, 30, 50, 70 seconds. Because the machine is a time-sharing system, we ran each experiment three times to ensure the accuracy of the measurement results; we saw virtually no variation in the results across the three experiments.

Figure 12 shows the complementary cumulative distribution of the execution time for each of the four epoch intervals. As shown in the graph, the execution of nearly every epoch was completed within the epoch interval. For example, the curve for a ten-second epoch interval shows that more than 99% of epochs could be processed within one second; however 0.1% of the epochs required more than ten seconds to complete. Our system occasionally lags behind the arrival of BGP updates, due to the bursty arrival pattern of BGP updates. Our data show that, while the average number of BGP updates per second is well below 100 (which corresponds to about 30 Kbps data rate), the maximum number of BGP updates received in our system in one second could well exceed 10,000 (which corresponds to 3 Mbps data rate).

Despite the existence of execution lags, for an epoch interval of 30 seconds, its percentage becomes much smaller (0.01%) by smoothing the BGP update bursts with a longer interval. The execution lag is completely eliminated when we set the epoch interval to 70 seconds; that is, every interval of 70 seconds worth of BGP updates was completely processed in less than 70 seconds. We believe the occasional execution lag is acceptable. Recall that each event is identified only if at least a period of event timeout elapses after the arrival of the last BGP update in the event. Typically the timeout value is a few tens of seconds (70 seconds, in our experiments). That is, even with instantaneous processing, each BGP update would have to wait for at least 70 seconds before a report is generated for the network operators. As such, smoothing the processing of BGP updates over a few tens of seconds does not introduce a problem.

9 Related Work

There is a large body of literature on characterizing BGP data using passive monitoring [3, 4, 9, 22, 29] as well as active route injection [17]. Our study is also preceded by several recent efforts [6, 8, 13, 15, 30] to identify the location and cause of routing changes by analyzing BGP update messages along three dimensions: time, views, and prefixes. Our work is similar in that we analyze BGP data along the same dimensions to group related routing changes. However, we focus on organizing large volumes of BGP updates seen in a single AS in real time into a small number of reports belonging to categories directly useful to operators to help mitigate the problems.

In analyzing BGP data collected from multiple vantage points within a single AS, our work is similar to the BorderGuard [12] study that identifies inconsistent routing advertisements from peers. In contrast, we classify *all* routing changes seen by the border routers into useful categories. The work in [27] presents a strawman proposal where each AS collects BGP data from its border routers as part of an end-to-end service for identifying the location and cause of routing changes. Each AS uses the data to detect and explain its own *internal* routing changes, rather than trying to detect and diagnose inter-domain routing events. Recent work [23] has considered how to detect network anomalies through a joint analysis of traffic and routing data. This work looks for significant changes in both the volume of traffic and the number of update messages, without delving in to the details about the specific destination prefixes and event types involved.

10 Conclusion and Future Work

We have presented the design and evaluation of an online system for identifying important BGP routing changes in an IP network. Using the concise r-vector data structure to capture BGP routing changes, we identified five categories of BGP routing disruptions that vary in the severity of the impact on the traffic. Applying the tool to eight weeks of routing and traffic data from a tier-1 ISP network, we identified several ways for operators to improve the routing stability of the network. Despite having route-flap damping features enabled on all of the routers, our tool surprisingly discovered a large number of updates from persistently flapping prefixes and identified three causes. Meanwhile, we found that hot-potato routing changes and eBGP session resets were responsible for many of the large routing disruptions.

In our ongoing work, we are extending the system to

use fine-grained traffic data collected at the ingress points for more precise estimates of the traffic impact. We also plan to explore routing architectures, operational practices, and protocol enhancements that reduce the likelihood and impact of the routing disruptions associated with hot-potato changes and eBGP session resets. Finally, we plan to explore automated techniques for responding to disruptions by reconfiguring the routing protocols to improve network performance.

Acknowledgments

Thanks to Alex Gerber, Joel Gottlieb, Renata Teixeira, and Jen Yates for their help with the measurement data. Thanks also to Mike Bailey, Jay Borkenhagen, Jaideep Chandrashekar, Evan Cooke, Nick Feamster, Ningning Hu, Ramana Kompella, Dan Pei, Aman Shaikh, Kang Shin, and the anonymous reviewers for their detailed comments on an earlier version of the paper. We would also like to thank our shepherd, Ramesh Govindan.

References

[1] Netflow. `http://www.cisco.com/warp/public/732/Tech/nmp/netflow/index.shtml`.

[2] RIPE Routing-WG recommendations for coordinated route-flap damping parameters, Oct. 2001. Document ID: ripe-229, `http://www.ripe.net/ripe/docs/routeflap-damping.html`.

[3] C. LABOVITZ, R. MALAN, AND F. JAHANIAN. Internet routing instability. In *Proc. ACM SIGCOMM* (1997).

[4] C. LABOVITZ, R. MALAN, AND F. JAHANIAN. Origins of Internet routing instability. In *Proc. IEEE INFOCOM* (1999).

[5] C. VILLAMIZAR, R. CHANDRA, AND R. GOVINDAN. BGP route flap damping. *RFC 2439* (1998).

[6] CAESAR, M., SUBRAMANIAN, L., AND KATZ, R. H. Towards localizing root causes of BGP dynamics. Tech. Rep. CSD-03-1292, UC Berkeley, November 2003.

[7] CHANG, D., GOVINDAN, R., AND HEIDEMANN, J. An empirical study of router response to large BGP routing table load. In *Proc. Internet Measurement Workshop* (November 2002).

[8] CHANG, D.-F., GOVINDAN, R., AND HEIDEMANN, J. The temporal and topological characteristics of BGP path changes. In *Proc. IEEE ICNP* (November 2003).

[9] D. WETHERALL, R. MAHAJAN, AND T. ANDERSON. Understanding BGP misconfigurations. In *Proc. ACM SIGCOMM* (2002).

[10] E. CHEN AND T. BATES. An application of the BGP community attribute in multi-home routing. *RFC 1998* (1996).

[11] FANG, W., AND PETERSON, L. Inter-AS traffic patterns and their implications. In *Proc. IEEE Global Internet* (December 1999).

[12] FEAMSTER, N., MAO, Z. M., AND REXFORD, J. BorderGuard: Detecting cold potatoes from peers. In *Proc. Internet Measurement Conference* (October 2004).

[13] FELDMANN, A., MAENNEL, O., MAO, Z. M., BERGER, A., AND MAGGS, B. Locating Internet routing instabilities. In *Proc. ACM SIGCOMM* (August 2004).

[14] LABOVITZ, C., AHUJA, A., BOSE, A., AND JAHANIAN, F. Delayed Internet routing convergence. *IEEE/ACM Trans. on Networking 9*, 3 (June 2001), 293–306.

[15] LAD, M., NANAVATI, A., MASSEY, D., AND ZHANG, L. An algorithmic approach to identifying link failures. In *Proc. Pacific Rim Dependable Computing* (2004).

[16] LONVICK, C. The BSD syslog protocol. RFC 3164, August 2001.

[17] MAO, Z. M., BUSH, R., GRIFFIN, T. G., AND ROUGHAN, M. BGP beacons. In *Proc. Internet Measurement Conference* (2003).

[18] MCPHERSON, D., GILL, V., WALTON, D., AND RETANA, A. Border gateway protocol (BGP) persistent route oscillation condition. RFC 3345, August 2002.

[19] N. DUFFIELD, C. LUND, AND M. THORUP. Estimating flow distributions from sampled flow statistics. In *Proc. ACM SIGCOMM* (2003).

[20] O. MAENNEL AND A. FELDMANN. Realistic BGP traffic for test labs. In *Proc. ACM SIGCOMM* (2002).

[21] REKHTER, Y., LI, T., AND HARES, S. A Border Gateway Protocol 4 (BGP-4). Internet Draft draft-ietf-idr-bgp4-26.txt, work in progress, Oct. 2004.

[22] REXFORD, J., WANG, J., XIAO, Z., AND ZHANG, Y. BGP routing stability of popular destinations. In *Proc. Internet Measurement Workshop* (2002).

[23] ROUGHAN, M., GRIFFIN, T., MAO, Z. M., GREENBERG, A., AND FREEMAN, B. Combining routing and traffic data for detection of IP forwarding anomalies. In *Proc. SIGCOMM Network Troubleshooting Workshop* (2004).

[24] ROUTEVIEWS. http://www.routeviews.org/.

[25] T. GRIFFIN AND G. WILFONG. Analysis of the MED Oscillation problem in BGP. In *Proc. IEEE ICNP* (November 2002).

[26] TEIXEIRA, R., DUFFIELD, N., REXFORD, J., AND ROUGHAN, M. Traffic matrix reloaded: Impact of routing changes. In *Proc. Passive and Active Measurement Workshop* (March/April 2005).

[27] TEIXEIRA, R., AND REXFORD, J. A measurement framework for pin-pointing routing changes. In *Proc. SIGCOMM Network Troubleshooting Workshop* (September 2004).

[28] TEIXEIRA, R., SHAIKH, A., GRIFFIN, T., AND REXFORD, J. Dynamics of hot-potato routing in IP networks. In *Proc. ACM SIGMETRICS* (2004).

[29] WANG, L., ZHAO, X., PEI, D., BUSH, R., MASSEY, D., MANKIN, A., WU, S. F., AND ZHANG, L. Observation and analysis of BGP behavior under stress. In *Proc. Internet Measurement Workshop* (2002).

[30] WONG, T., JACOBSON, V., AND ALAETTINOGLU, C. Making sense of BGP, Feb. 2004. NANOG presentation.

Notes

[1] Since the routes from $c1$ and $c3$ have the same IGP path cost, the router performs an arbitrary tiebreak in the last step in Table 1.

[2] This would require either (i) extending today's commercial routers to provide a feed of updates for each eBGP session or (ii) deploying packet monitors on all peering links to capture BGP update messages.

[3] The requirement of having at least one eBGP-learned route for the prefix is necessary to distinguish the "multiple external disruption" category from the "loss/gain of reachability" category.

[4] Our system can generate reports about large clusters (*i.e.,* when the count of events exceeds a threshold) before the 660-second timer expires, to allow operators to react more quickly to significant events.

[5] In theory, we could check that the number drops to zero. However, maintaining the prefix count for each session in real time (*e.g.,* updating whenever a prefix is withdrawn or advertised) introduces substantial overhead. Instead, noticing that the prefix count stays stable, we sample the count every two hours and maintain a moving average. We flag cases where the number of events in the cluster is more than 90% of the average number of prefixes associated with that session.

[6] Furthermore, syslog uses UDP as its transport layer and therefore its data can be lost during delivery.

Design and Implementation of a Routing Control Platform

Matthew Caesar, *UC Berkeley*
Donald Caldwell, *AT&T Labs-Research*
Nick Feamster, *MIT*
Jennifer Rexford, *Princeton University*
Aman Shaikh, *AT&T Labs-Research*
Jacobus van der Merwe, *AT&T Labs-Research*

Abstract

The routers in an Autonomous System (AS) must distribute the information they learn about how to reach external destinations. Unfortunately, today's internal Border Gateway Protocol (iBGP) architectures have serious problems: a "full mesh" iBGP configuration does not scale to large networks and "route reflection" can introduce problems such as protocol oscillations and persistent loops. Instead, we argue that a Routing Control Platform (RCP) should collect information about external destinations and internal topology and select the BGP routes for each router in an AS. RCP is a logically-centralized platform, separate from the IP forwarding plane, that performs route selection on behalf of routers and communicates selected routes to the routers using the unmodified iBGP protocol. RCP provides scalability without sacrificing correctness. In this paper, we present the design and implementation of an RCP prototype on commodity hardware. Using traces of BGP and internal routing data from a Tier-1 backbone, we demonstrate that RCP is fast and reliable enough to drive the BGP routing decisions for a large network. We show that RCP assigns routes correctly, even when the functionality is replicated and distributed, and that networks using RCP can expect comparable convergence delays to those using today's iBGP architectures.

1 Introduction

The Border Gateway Protocol (BGP), the Internet's interdomain routing protocol, is prone to protocol oscillation and forwarding loops, highly sensitive to topology changes inside an Autonomous System (AS), and difficult for operators to understand and manage. We address these problems by introducing a Routing Control Platform (RCP) that computes the BGP routes for each router in an AS based on complete routing information and higher-level network engineering goals [1, 2]. This paper describes the design and implementation of an RCP prototype that is fast and reliable enough to coordinate routing for a large backbone network.

1.1 Route Distribution Inside an AS

The routers in a single AS exchange routes to external destinations using a protocol called internal BGP (iBGP). Small networks are typically configured as a "full mesh" iBGP topology, with an iBGP session between each pair of routers. However, a full-mesh configuration does not scale because each router must: (i) have an iBGP session with every other router, (ii) send BGP update messages to every other router, (iii) store a local copy of the advertisements sent by each neighbor for each destination prefix, and (iv) have a new iBGP session configured whenever a new router is added to the network. Although having a faster processor and more memory on every router would support larger full-mesh configurations, the installed base of routers lags behind the technology curve, and upgrading routers is costly. In addition, BGP-speaking routers do not always degrade gracefully when their resource limitations are reached; for example, routers crashing or experiencing persistent routing instability under such conditions have been reported [3]. In this paper, we present the design, implementation, and evaluation of a solution that behaves like a full-mesh iBGP configuration with much less overhead and no changes to the installed base of routers.

To avoid the scaling problems of a full mesh, today's large networks typically configure iBGP as a hierarchy of *route reflectors* [4]. A route reflector selects a single BGP route for each destination prefix and advertises the route to its clients. Adding a new router to the system simply requires configuring iBGP sessions to the router's route reflector(s). Using route reflectors reduces the memory and connection overhead on the routers, at the expense of compromising the behavior of the underlying network. In particular, a route reflector does not necessarily select

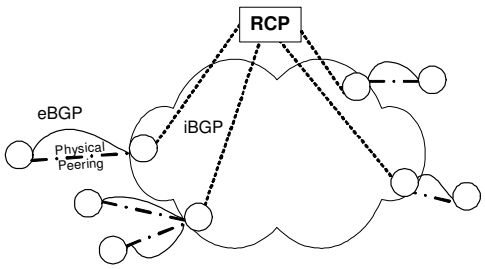


Figure 1: Routing Control Platform (RCP) in an AS

the same BGP route that its clients would have chosen in a full-mesh configuration. Unfortunately, the routers along a path through the AS may be assigned different BGP routes from different route reflectors, leading to inconsistencies [5]. These inconsistencies can cause protocol oscillation [6, 7, 8] and persistent forwarding loops [6]. To prevent these problems, operators must ensure that route reflectors and their clients have a consistent view of the internal topology, which requires configuring a large number of routers as route reflectors. This forces large backbone networks to have dozens of route reflectors to reduce the likelihood of inconsistencies.

1.2 Routing Control Platform (RCP)

RCP provides both the intrinsic correctness of a full-mesh iBGP configuration and the scalability benefits of route reflectors. RCP selects BGP routes on behalf of the routers in an AS using a complete view of the available routes and IGP topology. As shown in Figure 1, RCP has iBGP sessions with each of the routers; these sessions allow RCP to learn BGP routes and to send each router a routing decision for each destination prefix. Unlike a route reflector, RCP may send a different BGP route to each router. This flexibility allows RCP to assign each router the route that it would have selected in a full-mesh configuration, while making the number of iBGP sessions at each router independent of the size of the network. We envision that RCP may ultimately exchange interdomain routing information with neighboring domains, while still using iBGP to communicate with its own routers. Using the RCP to exchange reachability information across domains would enable the Internet's routing architecture to evolve [1].

To be a viable alternative to today's iBGP solutions, RCP must satisfy two main design goals: (i) consistent assignment of routes even when the functionality is replicated and distributed for reliability and (ii) fast response to network events, such as link failures and external BGP routing changes, even when computing routes for a large number of destination prefixes and routers. This paper demonstrates that RCP can be made fast and reliable enough to supplant today's iBGP architectures, without requiring any changes to the implementation of the legacy routers. After a brief overview of BGP routing in Section 2, Section 3 presents the RCP architecture and describes how to compute consistent forwarding paths, without requiring any explicit coordination between the replicas. In Section 4, we describe a prototype implementation, built on commodity hardware, that can compute and disseminate routing decisions for a network with hundreds of routers. Section 5 demonstrates the effectiveness of our prototype by replaying BGP and OSPF messages from a large backbone network; we also discuss the challenges of handling OSPF-induced BGP routing changes and evaluate one potential solution. Section 6 summarizes the contributions of the paper.

1.3 Related Work

We extend previous work on route monitoring [9, 10] by building a system that also *controls* the BGP routing decisions for a network. In addition, RCP relates to recent work on router software [11, 12, 13], including the proprietary systems used in today's commercial routers; in contrast to these efforts, RCP makes per-router routing decisions for an *entire network*, rather than a single router. Our work relates to earlier work on applying routing policy at route servers at the exchange points [14], to obviate the need for a full mesh of eBGP sessions; in contrast, RCP focuses on improving the scalability and correctness of distributing and selecting BGP routes within a single AS. The techniques used by the RCP for efficient storage of the per-router routes are similar to those employed in route-server implementations [15].

Previous work has proposed changes to iBGP that prevent oscillations [16, 7]; unlike RCP, these other proposals require significant modifications to BGP-speaking routers. RCP's logic for determining the BGP routes for each router relates to previous research on network-wide routing models for traffic engineering [17, 18]; RCP focuses on *real-time control* of BGP routes rather than modeling the BGP routes in today's routing system. Previous work has highlighted the need for a system that has network-wide control of BGP routing [1, 2]; in this paper, we present the design, implementation, and evaluation of such a system. For an overview of *architecture* and *standards* activities on separating routing from routers, see the related work discussions in [1, 2].

2 Interoperating With Existing Routers

This section presents an overview of BGP routing inside an AS and highlights the implications on how RCP must work to avoid requiring changes to the installed base of IP routers.

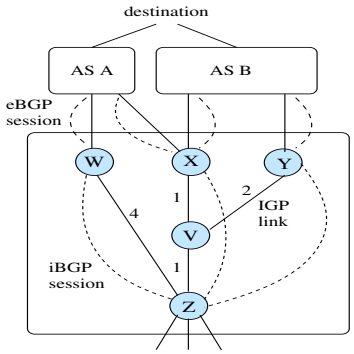


Figure 2: Network with three egress routers connecting to two neighboring ASes: Solid lines correspond to physical links (annotated with IGP link weights) and dashed lines correspond to BGP sessions.

0. Ignore if egress router unreachable
1. Highest local preference
2. Lowest AS path length
3. Lowest origin type
4. Lowest MED (with *same* next-hop AS)
5. eBGP-learned over iBGP-learned
6. Lowest IGP path cost to egress router
7. Lowest router ID of BGP speaker

Table 1: Steps in the BGP route-selection process

Partitioning of functionality across routing protocols: In most backbone networks, the routers participate in three different routing protocols: external Border Gateway Protocol (eBGP) to exchange reachability information with neighboring domains, internal BGP (iBGP) to propagate the information inside the AS, and an Interior Gateway Protocol (IGP) to learn how to reach other routers in the same AS, as shown in Figure 2. BGP is a path-vector protocol where each network adds its own AS number to the path before propagating the announcement to the next domain; in contrast, IGPs such as OSPF and IS-IS are typically link-state protocols with a tunable weight on each link. Each router combines the information from the routing protocols to construct a local forwarding table that maps each destination prefix to the next link in the path. *In our design, RCP assumes responsibility for assigning a single best BGP route for each prefix to each router and distributing the routes using iBGP, while relying on the routers to "merge" the BGP and IGP data to construct their forwarding tables.*

BGP route-selection process: To select a route for each prefix, each router applies the decision process in Table 1 to the set of routes learned from its eBGP and iBGP neighbors [19]. The decision process essentially compares the routes based on their many attributes. In the simplest case, a router selects the route with the shortest AS path (step 2), breaking a tie based on the ID of the router who advertised the route (step 7). However, other steps depend on route attributes, such as local preference, that are assigned by the routing policies configured on the border routers. *RCP must deal with the fact that the border routers apply policies to the routes learned from their eBGP neighbors and all routers apply the route-selection process to the BGP routes they learn.*

Selecting the closest egress router: In backbone networks, a router often has multiple BGP routes that are "equally good" through step 5 of the decision process. For example, router Z in Figure 2 learns routes to the destination with the same AS path length from three border routers W, X, and Y. To reduce network resource consumption, the BGP decision process at each router selects the route with the *closest* egress router, in terms of the IGP path costs. Router Z selects the BGP route learned from router X with an IGP path cost of 2. This practice is known as "early-exit" or "hot-potato" routing. *RCP must have a real-time view of the IGP topology to select the closest egress router for each destination prefix on behalf of each router. When the IGP topology changes, RCP must identify which routers should change the egress router they are using.*

Challenges introduced by hot-potato routing: A single IGP topology change may cause multiple routers to change their BGP routing decisions for multiple prefixes. If the IGP weight of link V–X in Figure 2 increased from 1 to 3, then router Z would start directing traffic through egress Y instead of X. When multiple destination prefixes are affected, these hot-potato routing changes can lead to large, unpredictable shifts in traffic [20]. In addition, the network may experience long convergence delays because of the overhead on the routers to revisit the BGP routing decisions across many prefixes. Delays of one to two minutes are not uncommon [20]. *To implement hot-potato routing, RCP must determine the influence of an IGP change on every router for every prefix. Ultimately, we view RCP as a way to move beyond hot-potato routing toward more flexible ways to select egress routers, as discussed in Section 5.4.*

3 RCP Architecture

In this section, we describe the RCP architecture. We first present the three building blocks of the RCP: the IGP Viewer, the BGP Engine, and the Route Control Server (RCS). We describe the information that is available to each module, as well as the constraints that the RCS must satisfy when assigning routes. We then discuss how RCP's functionality can be replicated and distributed across many physical nodes in an AS while maintaining consistency and correctness. Our analysis shows that there is no need for the replicas to run a separate consistency protocol: since the RCP is designed such that each RCS replica makes routing decisions only for the partitions for which it has complete IGP topology

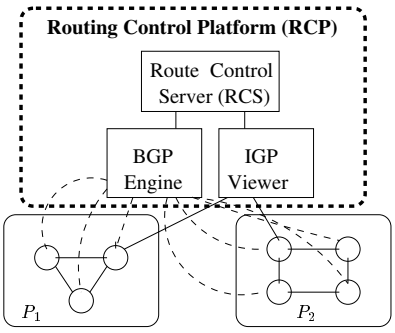


Figure 3: RCP interacts with the routers using standard routing protocols. RCP obtains IGP topology information by establishing IGP adjacencies (shown with solid lines) with one or more routers in the AS and BGP routes via iBGP sessions with each router (shown with dashed lines). RCP can control and obtain routing information from routers in separate network partitions (P_1 and P_2). Although this figure shows RCP as a single box, the functionality can be replicated and distributed, as we describe in Section 3.2.

and BGP routes, every replica will make the same routing assignments, even without a consistency protocol.

3.1 RCP Modules

To compute the routes that each router would have selected in a "full mesh" iBGP configuration, RCP must obtain both the IGP topology information and the best route to the destination from every router that learns a route from neighboring ASes. As such, RCP comprises of three modules: the IGP Viewer, the BGP Engine, and the Route Control Server. The *IGP Viewer* establishes IGP adjacencies to one or more routers, which allows the RCP to receive IGP topology information. The *BGP Engine* learns BGP routes from the routers and sends the RCS's route assignments to each router. The *Route Control Server (RCS)* then uses the IGP topology from the IGP Viewer information and the BGP routes from the BGP engine to compute the best BGP route for each router.

RCP communicates with the routers in an AS using standard routing protocols, as summarized in Figure 3. Suppose the routers R in a single AS form an IGP connectivity graph $G = (R, E)$, where E are the edges in the IGP topology. Although the IGP topology within an AS is typically a single connected component, failures of links, routers, or interfaces may occasionally create partitions. Thus, G contains one or more connected components; i.e., $G = \{P_1, P_2, \ldots, P_n\}$. The RCS only computes routes for partitions P_i for which it has complete IGP and BGP information, and it computes routes for each partition independently.

3.1.1 IGP Viewer

The RCP's *IGP Viewer* monitors the IGP topology and provides this information to the RCS. The IGP Viewer establishes IGP adjacencies to receive the IGP's link-state advertisements (LSAs). To ensure that the IGP Viewer never routes data packets, the links between the IGP Viewer and the routers should be configured with large IGP weights to ensure that the IGP Viewer is not an intermediate hop on any shortest path. Since IGPs such as OSPF and IS-IS perform *reliable flooding* of LSAs, the IGP Viewer maintains an up-to-date view of the IGP topology as the link weights change or equipment goes up and down. Use of flooding to disseminate LSAs implies that the IGP Viewer can receive LSAs from all routers in a partition by simply having an adjacency to a single router in that partition. This seemingly obvious property has an important implication:

Observation 1 *The IGP Viewer has the complete IGP topology for all partitions that it connects to.*

The IGP Viewer computes pairwise shortest paths for all routers in the AS and provides this information to the RCS. The IGP Viewer must discover only the path costs between any two routers in the AS, but it need not discover the weights of each IGP edge. The RCS then uses these path costs to determine, from any router in the AS, what the closest egress router should be for that router.

In some cases, a group of routers in the IGP graph all select the same router en route to one or more destinations. For example, a network may have a group of access routers in a city, all of which send packets out of that city towards one or more destinations via a single gateway router. These routers would always use the same BGP router as the gateway. These groups can be formed according to the IGP topology: for example, routers can be grouped according to OSPF "areas", since all routers in the same area typically make the same BGP routing decision. Because the IGP Viewer knows the IGP topology, it can determine which groups of routers should be assigned the same BGP route. By clustering routers in this fashion, the IGP Viewer can reduce the number of independent route computations that the RCS must perform. While IGP topology is a convenient way for the IGP Viewer to determine these groups of routers, the groups need not correspond to the IGP topology; for example, an operator could dictate the grouping.

3.1.2 BGP Engine

The *BGP Engine* maintains an iBGP session with each router in the AS. These iBGP sessions allow the RCP to (1) learn about candidate routes and (2) communicate its routing decisions to the routers. Since iBGP runs over

TCP, a BGP Engine need not be physically adjacent to every router. In fact, a BGP Engine can establish and maintain iBGP sessions with any router that is reachable via the IGP topology, which allows us to make the following observation:

Observation 2 *A BGP Engine can establish iBGP sessions to all routers in the IGP partitions that it connects to.*

Here, we make a reasonable assumption that IGP connectivity between two endpoints is sufficient to establish a BGP session between them; in reality, persistent congestion or misconfiguration could cause this assumption to be violated, but these two cases are anomalous. In practice, routers are often configured to place BGP packets in a high-priority queue in the forwarding path to ensure the delivery of these packets even during times of congestion.

In addition to receiving BGP updates, the RCP uses the iBGP sessions to send the chosen BGP routes to the routers. Because BGP updates have a "next hop" attribute, the BGP Engine can advertise BGP routes with "next hop" addresses of other routers in the network. This characteristic means that *the BGP Engine does not need to forward data packets*. The BGP routes typically carry "next hop" attributes according to the egress router at which they were learned. Thus, the RCS can send a route to a router with the next hop attribute unchanged, and routers will forward packets towards the egress router.

A router interacts with the BGP Engine in the same way as it would with a normal BGP-speaking router, but the BGP Engine can send a different route to each router. (In contrast, a traditional route reflector would send the same route to each of its neighboring routers.) A router only sends BGP update messages to the BGP Engine when selecting a new best route learned from a neighboring AS. Similarly, the BGP Engine only sends an update when a router's decision should change.

3.1.3 Route Control Server (RCS)

The RCS receives IGP topology information from the IGP Viewer and BGP routes from the BGP Engine, computes the routes for a group of routers, and returns the resulting route assignments to the routers using the BGP Engine. The RCS does not return a route assignment to any router that has already selected a route that is "better" than any of the other candidate routes, according to the decision process in Table 1. To make routing decisions for a group of routers in some partition, the following must be true:

Observation 3 *An RCS can only make routing decisions for routers in a partition for which it has* both *IGP and BGP routing information.*

Note that the previous observations guarantee that the RCS can (and will) make path assignments for all routers in that partition. Although the RCS has considerable flexibility in assigning routes to routers, one reasonable approach would be to have the RCS send to each router the route that it would have selected in a "full mesh" iBGP configuration. To emulate a full-mesh iBGP configuration, the RCS executes the BGP decision process in Table 1 on behalf of each router. The RCS can perform this computation because: (1) knowing the IGP topology, the RCS can determine the set of egress routers that are reachable from any router in the partitions that it sees; (2) the next four steps in the decision process compare attributes that appear in the BGP messages themselves; (3) for step 5, the RCS considers a route as eBGP-learned for the router that sent the route to the RCP, and as an iBGP-learned route for other routers; (4) for step 6, the RCS compares the IGP path costs sent by the IGP Viewer; and (5) for step 7, the RCS knows the router ID of each router because the BGP Engine has an iBGP session with each of them. After computing the routes, the RCS can send each router the appropriate route.

Using the high-level correctness properties from previous work as a guide [21], we recognize that routing within the network must satisfy the following properties (note that iBGP does *not* intrinsically satisfy them [6, 21]):

Route validity: The RCS should not assign routes that create forwarding loops, blackholes, or other anomalies that prevent packets from reaching their intended destinations. To satisfy this property, two invariants must hold. First, the RCS must assign routes such that the routers along the shortest IGP path from any router to its assigned egress router must be assigned a route with the same egress router. Second, the RCS must assign a BGP route such that the IGP path to the next-hop of the route only traverses routers in the same partition as the next-hop.

When the RCS computes the same route assignments as those the routers would select in a full mesh iBGP configuration, the first invariant will always hold, for the same reason that it holds in the case of full mesh iBGP configuration. In a full mesh, each router simply selects the egress router with the shortest IGP path. All routers along the shortest path to that egress also select the same closest egress router. The second invariant is satisfied because the RCS never assigns an egress router to a router in some other partition. Generally, the RCS has considerable flexibility in assigning paths; the RCS must guarantee that these properties hold even when it is not emu-

lating a full mesh configuration.

Path visibility: Every router should be able to exchange routes with at least one RCS. Each router in the AS should receive some route to an external destination, assuming one exists. To ensure that this property is satisfied, each partition must have at least one IGP Viewer, one BGP Engine, and one RCS. Replicating these modules reduces the likelihood that a group of routers is partitioned such that it cannot reach at least one instance of these three components. If the RCS is replicated, then two replicas may assign BGP routes to groups of routers along the same IGP path between a router and an egress. To guarantee that two replicas do not create forwarding loops when they assign routes to routers in the same partition, they must make consistent routing decisions. If a network has multiple RCSes, the route computation performed by the RCS must be deterministic: the same IGP topology and BGP route inputs must always produce the same outcome for the routers.

If a partition forms such that a router is partitioned from RCP, then we note that (1) the situation is no worse than today's scenario, when a router cannot receive BGP routes from its route reflector and (2) in many cases, the router will still be able to route packets using the routes it learns via eBGP, which will likely be its best routes since it is partitioned from most of the remaining network anyway.

3.2 Consistency with Distributed RCP

In this section, we discuss the potential consistency problems introduced by replicating and distributing the RCP modules. To be robust to network partitions and avoid creating a single point of failure, the RCP modules should be replicated. (We expect that many possible design strategies will emerge for assigning routers to replicas. Possible schemes include using the closest replica, having primary and backup replicas, etc.) Replication introduces the possibility that each RCS replica may have different views of the network state (i.e., the IGP topology and BGP routes). These inconsistencies may be either transient or persistent and could create problems such as routing loops if routers were learning routes from different replicas.[1] The potential for these inconsistencies would seem to create the need for a consistency protocol to ensure that each RCS replica has the same view of the network state (and, thus, make consistent routing decisions). In this section, we discuss the nature and consequences of these inconsistencies and present the surprising result that no consistency protocol is required to prevent persistent inconsistencies.

After discussing why we are primarily concerned with consistency of the RCS replicas in steady state, we explain how our replication strategy guarantees that the

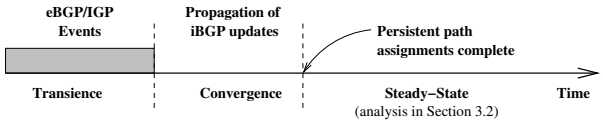


Figure 4: Periods during convergence to steady state for a single destination. Routes to a destination within an AS are stable most of the time, with periods of transience (caused by IGP or eBGP updates). Rather than addressing the behavior during the transient period, we analyze the consistency of paths assigned during steady state.

RCS replicas make the same routing decisions for each router in the steady state. Specifically, we show that, if multiple RCS replicas have IGP connectivity to some router in the AS, then those replicas will all make the same path assignment for that router. We focus our analysis on the consistency of RCS path assignments in steady state (as shown in Figure 4).

3.2.1 Transient vs. Persistent Inconsistencies

Since each replica may receive BGP and IGP updates at different times, the replicas may not have the same view of the routes to every destination at any given time; as a result, each replica may make different routing decisions for the same set of routers. Figure 4 illustrates a timeline that shows this transient period. During transient periods, routes may be inconsistent. On a per-prefix basis, long transient periods are *not the common case*: although BGP update traffic is fairly continuous, the update traffic *for a single destination* as seen by a single AS is relatively bursty, with prolonged periods of silence. That is, a group of updates may arrive at several routers in an AS during a relatively short time interval (i.e., seconds to minutes), but, on longer timescales (i.e., hours), the BGP routes for external destinations are relatively stable [22].

We are concerned with the consistency of routes for each destination *after* the transient period has ended. Because the network may actually be partitioned in "steady state", the RCP must still consider network partitions that may exist during these periods. Note that *any intra-AS routing protocol, including any iBGP configuration, will temporarily have inconsistent path assignments when BGP and IGP routes are changing continually*. Comparing the nature and extent of these transient inconsistencies in RCP to those that occur under a typical iBGP configuration is an area for future work.

3.2.2 RCP Replicas are Consistent in Steady State

The RCS replicas should make consistent routing decisions in steady state. Although it might seem that such a consistency requirement mandates a separate consistency protocol, we show in this section that such a protocol is not necessary.

Proposition 1 *If multiple RCSes assign paths to routers in P_i, then each router in P_i would receive the same route assignment from each RCS.*

Proof. Recall that two RCSes will only make different assignments to a router in some partition P_i if the replicas receive different inputs (i.e., as a result of having BGP routes from different groups of routers or different views of IGP topology). Suppose that RCSes A and B both assign routes to some router in P_i. By Observation 1, both RCSes A and B must have IGP topology information for *all* routers in P_i, and from Observation 2, they also have complete BGP routing information. It follows from Observation 3 that both RCSes A and B can make route assignments for all routers in P_i. Furthermore, since both RCSes have complete IGP and BGP information for the routers in P_i (i.e., the replicas receive the same inputs), then RCSes A and B will make the same route assignment to each router in P_i. ■

We note that certain failure scenarios may violate Observation 2; there may be circumstances under which IGP-level connectivity exists between the BGP engine and some router but, for some reason, the iBGP session fails (e.g., due to congestion, misconfiguration, software failure, etc.) As a result, Observation 3 may be overly conservative, because there may exist routers in some partition for which two RCSes may have BGP routing information from different subsets of routers in that partition. If this is the case, then, by design, neither RCS will assign routes to *any* routers in this partition, even though, collectively, both RCSes have complete BGP routing information. In this case, not having a consistency protocol affects liveness, but not correctness—in other words, two or more RCSes may fail to assign routes to routers in some partition even when they collectively have complete routing information, but in no case will two or more RCSes assign different routes to the same router.

4 RCP Architecture and Implementation

To demonstrate the feasibility of the RCP architecture, this section presents the design and implementation of an RCP prototype. Scalability and efficiency pose the main challenges, because backbone ASes typically have many routers (e.g., 500–1000) and destination prefixes (e.g., 150,000–200,000), and the routing protocols must converge quickly. First, we describe how the RCS computes the BGP routes for each group of routers in response to BGP and IGP routing changes. We then explain how the IGP Viewer obtains a view of the IGP topology and provides the RCS with only the necessary information for computing BGP routes. Our prototype of the IGP Viewer is implemented for OSPF; when describing our prototype, we will describe the IGP Viewer as the "OSPF Viewer". Finally, we describe how the BGP Engine exchanges BGP routing information with the routers in the AS and the RCS.

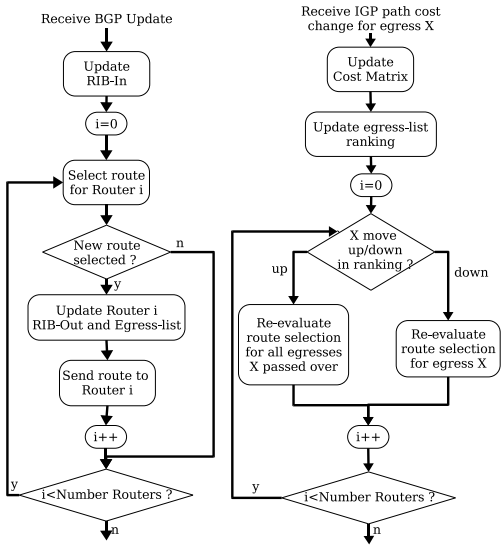


Figure 5: Route Control Server (RCS) functionality

4.1 Route Control Server (RCS)

The RCS processes messages received from both the BGP Engine(s) and the OSPF Viewer(s). Figure 5 shows the high level processing performed by the RCS. The RCS receives update messages from the BGP Engine(s) and stores the incoming routes in a Routing Information Base (RIB). The RCS perform per-router route selection and stores the selected routes in a per-router RIB-Out. The RIB-In and RIB-Out tables are implemented as a trie indexed on prefix. The RIB-In maintains a list of routes learned for each prefix; each BGP route has a "next hop" attribute that uniquely identifies the egress router where the route was learned. As shown in Figure 5, the RCS also receives the IGP path cost for each pair of routers from the IGP Viewer. The RCS uses the RIB-In to compute the best BGP routes for each router, using the IGP path costs in steps 0 and 6 of Table 1. After computing a route assignment for a router, the RCS sends that route assignment to the BGP Engine, which sends the update message to the router. The path cost changes received from the OSPF Viewer might require the RCS to re-compute selected routes when step 6 in the BGP decision process was used to select a route and the path cost to the selected egress router changes. Finding the routes that are affected can be an expensive process and as shown in Figure 5, our design uses a path-cost based ranking of egress routers to perform this efficiently. We now describe this approach and other design insights in

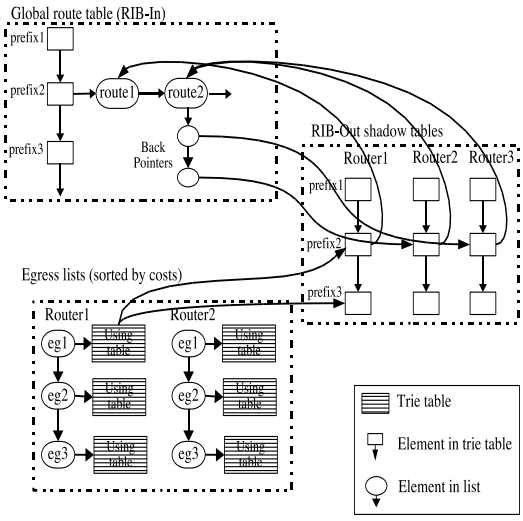


Figure 6: RCS RIB-In and RIB-Out data structures and egress lists

more detail with the aid of Figure 6, which shows the main RCS data structures:

Store only a single copy of each BGP route. Storing a separate copy of each router's BGP routes for every destination prefix would require an extraordinary amount of memory. To reduce storage requirements, the RCS only stores routes in the RIB-In table. The "next hop" attribute of the BGP route uniquely identifies the egress router where the BGP route was learned. Upon receiving an update message, the RCS can index the RIB-In by prefix and can add, update, or remove the appropriate route based on the next-hop attribute. To implement the RIB-Out, the RCS employs per-router *shadow tables* as a prefix-indexed trie containing pointers to the RIB-In table. Figure 6 shows two examples of these pointers from the RIB-Out to the RIB-In: router1 has been assigned the route1 for prefix2, whereas router2 and router3 have both been assigned route2 for prefix2.

Keep track of the routers that have been assigned each route. When a route is withdrawn, the RCS must recompute the route assignment for any router that was using the withdrawn route. To quickly identify the affected routers, each route stored in the RIB-In table includes a list of back pointers to the routers assigned this route. For example, Figure 6 shows two pointers from route2 in the RIB-In for prefix2 to indicate that router2 and router3 have been assigned this route. Upon receiving a withdrawal of the prefix from this next-hop attribute, the RCS reruns the decision process for each router in this list, with the remaining routes in the RIB-In, for those routers and prefix. Unfortunately, this ME optimization cannot be used for BGP *announcements*, because when a new route arrives, the RCS must recompute the route assignment for each router[2].

Maintain a ranking of egress routers for each router based on IGP path cost. A single IGP path-cost change may affect the BGP decisions for many destination prefixes at the ingress router. To avoid revisiting the routing decision for every prefix and router, the RCS maintains a ranking of egress points for each router sorted by the IGP path cost to the egress point (the "Egress lists" table in Figure 6). For each egress, the RCS stores pointers to the prefixes and routes in the RIB-Out that use the egress point (the "using table"). For example, router1 uses eg1 to reach both prefix2 and prefix3, and its using table contains pointers to those entries in the RIB-Out for router1 (which in turn point to the routes stored in the RIB-In). If the IGP path cost from router1 to eg1 increases, the RCS moves eg1 down the egress list until it encounters an egress router with a higher IGP path cost. The RCS then only recomputes BGP decisions for the prefixes that previously had been assigned the BGP route from eg1 (i.e., the prefixes contained in the using table). Similarly, if a path-cost change causes eg3 to become router1's closest egress point, the RCS resorts the egress list (moving eg3 to the top of the list) and only recomputes the routes for prefixes associated with the egresses routers "passed over" in the sorting process, i.e., eg1 and eg2, since they may now need to be assigned to eg3.

Assign routes to groups of related routers. Rather than computing BGP routes for each router, the RCS can assign the same BGP route for a destination prefix to a *group* of routers. These groups can be identified by the IGP Viewer or explicitly configured by the network operator. When the RCS uses groups, the RIB-Out and Egress-lists tables have entries for each *group* rather than each router, leading to a substantial reduction in storage and CPU overhead. The RCS also maintains a list of the routers in each group to instruct the BGP Engine to send the BGP routes to each member of the group. Groups introduce a trade-off between the desire to reduce overhead and the flexibility to assign different routes to routers in the same group. In our prototype implementation, we use the Points-of-Presence (which correspond to OSPF areas) to form the groups, essentially treating each POP as a single "node" in the graph when making BGP routing decisions.

4.2 IGP Viewer Instance: OSPF Viewer

The OSPF Viewer connects to one or more routers in the network to receive link-state advertisements (LSAs), as shown in Figure 3. The OSPF Viewer maintains an up-to-date view of the network topology and computes the path cost for each pair of routers. Figure 7 shows an overview of the processing performed by the OSPF Viewer. By providing path-cost changes and group membership information, the OSPF Viewer offloads work from the RCS in two main ways:

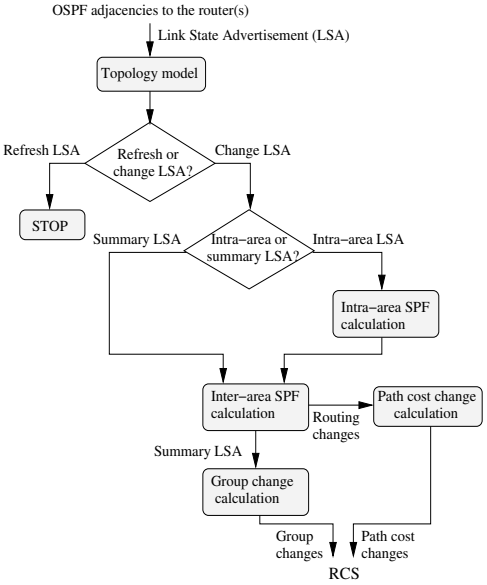


Figure 7: LSA Processing in OSPF Viewer

Send only path-cost changes to the RCS. In addition to originating an LSA upon a network change, OSPF periodically refreshes LSAs even if the network is stable. The OSPF Viewer filters the refresh LSAs since they do not require any action from the RCS. The OSPF Viewer does so by maintaining the network state as a topology model [9], and uses the model to determine whether a newly received LSA indicates a change in the network topology, or is merely a refresh as shown in Figure 7. For a change LSA, the OSPF Viewer runs shortest-path first (SPF) calculations from each router's viewpoint to determine the new path costs. Rather than sending all path costs to the RCS, the OSPF Viewer only passes the path costs that changed as determined by the "path cost change calculation" stage.

The OSPF Viewer must capture the influence of OSPF areas on the path costs. For scalability purposes, an OSPF domain may be divided into areas to form a hub-and-spoke topology. Area 0, known as the *backbone area*, forms the hub and provides connectivity to the non-backbone areas that form the spokes. Each link belongs to exactly one area. The routers that have links to multiple areas are called *border routers*. A router learns the entire topology of the area it has links into through "intra-area" LSAs. However, it does not learn the entire topology of remote areas (i.e., the areas in which the router does not have links), but instead learns the total cost of the paths to every node in remote areas from each border router the area has through "summary" LSAs.

It may seem that the OSPF Viewer can perform the SPF calculation over the entire topology, ignoring area boundaries. However, OSPF mandates that if two routers belong to the same area, the path between them must stay within the area even if a shorter path exists that traverses multiple areas. As such, the OSPF Viewer cannot ignore area boundaries while performing the calculation, and instead has to perform the calculation in two stages. In the first stage, termed the *intra-area stage*, the viewer computes path costs for each area separately using the intra-area LSAs as shown in Figure 7. Subsequently, the OSPF Viewer computes path costs between routers in different areas by combining paths from individual areas. We will term this stage of the SPF calculation as the *inter-area stage*. In some circumstances, the OSPF Viewer knows the topology of only a subset of areas, and not all areas. In this case, the OSPF Viewer can perform intra-area stage calculations only for the visible areas. However, use of summary LSAs from the border routers allows the OSPF Viewer to determine path costs to routers in non-visible areas from routers in visible areas during inter-area stage.

Reduce overhead at the RCS by combining routers into groups. The OSPF Viewer can capitalize on the area structure to reduce the number of routers the RCS must consider. To achieve this, the OSPF Viewer: (i) provides path cost information for all area 0 routers (which also includes border routers in non-zero areas), and (ii) forms a group of routers for each non-zero area and provides this group information. As an added benefit, the OSPF Viewer does not need physical connections to non-zero areas, since the summary LSAs from area 0 allows it to compute path costs from every area 0 router to every other router. The OSPF Viewer also uses the summary LSAs to determine the groups of routers. It is important to note that combining routers into groups is a construct internal to the RCP to improve efficiency, and it does not require any protocol or configuration changes in the routers.

4.3 BGP Engine

The BGP Engine receives BGP messages from the routers and sends them to the RCS. The BGP Engine also receives instructions from the RCS to send BGP routes to individual routers. We have implemented the BGP Engine by modifying the Quagga [11] software router to store the outbound routes on a *per-router* basis and accept route assignments from the RCS rather than computing the route assignments itself. The BGP Engine offloads work from the RCS by applying the following two design insights:

Cache BGP routes for efficient refreshes. The BGP Engine stores a local cache of the RIB-In and RIB-Out. The RIB-In cache allows the BGP Engine to provide the RCS with a fresh copy of the routes without affecting the routers, which makes it easy to introduce a new RCS replica or to recover from an RCS failure. Similarly, the RIB-Out cache allows the BGP Engine to re-send BGP

route assignments to operational routers without affecting the RCS, which is useful for recovering from the temporary loss of iBGP connectivity to the router. Because routes are assigned on a per-router basis, the BGP Engine maintains a RIB-Out for each router, using the same kind of data structure as the RCS.

Manage the low-level communication with the routers. The BGP Engine provides a simple, stable layer that interacts with the routers and maintains BGP sessions with the routers and multiplexes the update messages into a single stream to and from the RCS. It manages a large number of TCP connections and supports the low-level details of establishing BGP sessions and exchanging updates with the routers.

5 Evaluation

In this section, we evaluate our prototype implementation, with an emphasis on the scalability and efficiency of the system. The purpose of the evaluation is twofold. First, to determine the feasible operating conditions for our prototype, i.e., its performance as a function of the number of prefixes and routes, and the number of routers or router groups. Second, we want to determine what the bottlenecks (if any), would require further enhancements. We present our methodology in Section 5.1 and the evaluation results in Sections 5.2 and 5.3. In Section 5.4 we present experimental results of an approach that weakens the current tight coupling between IGP path-cost changes and BGP decision making.

5.1 Methodology

For a realistic evaluation, we use BGP and OSPF data collected from a Tier-1 ISP backbone on August 1, 2004. The BGP data contains both timestamped BGP updates as well as periodic table dumps from the network[3]. Similarly, the OSPF data contains timestamped Link State Advertisements (LSAs). We developed a *router-emulator* tool that reads the timestamped BGP and OSPF data and then "plays back" these messages against instrumented implementations of the RCP components. To initialize the RCS to realistic conditions, the router-emulator reads and replays the BGP table dumps before any experiments are conducted.

By selectively filtering the data, we use this single data set to consider the impact of network size (i.e., the number of routers or router groups in the network) and number of routes (i.e., the number of prefixes for which routes were received). We vary the network size by only calculating routes for a subset of the router groups in the network. Similarly, we only consider a subset of the prefixes to evaluate the impact of the number of routes on the RCP. Considering a subset of routes is relevant for networks that do not have to use a full set of Internet routes but might still benefit from the RCP functionality, such as private or virtual private networks.

For the RCS evaluation, the key metrics of interest are (i) the time taken to perform customized per-router route selection under different conditions and (ii) the memory required to maintain the various data structures. We measure these metrics in three ways:

- *Whitebox:* First, we perform whitebox testing by instrumenting specific RCS functions and measuring on the RCS both the memory usage and the time required to perform route selection when BGP and OSPF related messages are being processed.
- *Blackbox no queuing:* For blackbox no queuing, the router-emulator replays one message at a time and waits to see a response before sending the next message. This technique measures the additional overhead of the message passing protocol needed to communicate with the RCS.
- *Blackbox real-time:* For blackbox real-time testing, the router-emulator replays messages based on the timestamps recorded in the data. In this case, ongoing processing on the RCS can cause messages to be queued, thus increasing the effective processing times as measured at the router-emulator.

For all blackbox tests, the RCS sends routes back to the router-emulator to allow measurements to be done.

In Section 5.2, we focus our evaluation on how the RCP processes BGP updates and performs customized route selection. Our BGP Engine implementation extends the Quagga BGP daemon process and as such inherits many of its qualities from Quagga. Since we made no enhancements to the BGP protocol part of the BGP Engine but rely on the Quagga implementation we do not present an evaluation of its scalability in this paper[4]. Our main enhancement, the shadow tables maintained to realize per-router RIB-Outs, use the same data structures as the RCS, and hence, the evaluation of the RCS memory requirements is sufficient to show its feasibility.

In Section 5.3, we present an evaluation of the OSPF Viewer and the OSPF-related processing in the RCS. We evaluate the OSPF Viewer by having it read and process LSAs that were previously dumped to a file by a monitoring process. The whitebox performance of the OSPF Viewer is determined by measuring the time it takes to calculate the all pairs shortest paths and OSPF groups. The OSPF Viewer can also be executed in a test mode where it can log the path cost changes and group changes that would be passed to the RCS under normal operating conditions. The router-emulator reads and then plays back these logs against the RCS for blackbox evaluation of the RCS OSPF processing.

The evaluations were performed with the RCS and OSPF Viewer running on a dual 3.2 GHz Pentium-4 processor Intel system with 8 GB of memory and running a Linux 2.6.5 kernel. We ran the router-emulator on a 1 GHz Pentium-3 Intel system with 1 GB of memory and running a Linux 2.4.22 kernel.

5.2 BGP Processing

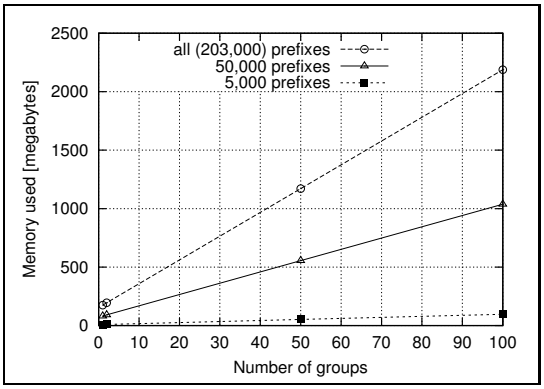


Figure 8: *Memory*: Memory used for varying numbers of prefixes.

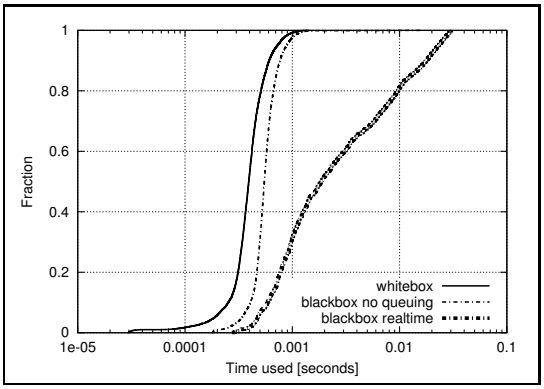


Figure 9: *Decision time, BGP updates*: RCS route selection time for whitebox testing (instrumented RCS), blackbox testing no queuing (single BGP announcements sent to RCS at a time), blackbox testing real-time (BGP announcements sent to RCS in real-time)

Figure 8 shows the amount of memory required by the RCS as a function of group size and for different numbers of prefixes. Recall that a group is a set of routers that would be receiving the same routes from the RCS. Backbone network topologies are typically built with a core set of backbone routers that interconnect points-of-presence (POPs), which in turn contain access routers [23]. All access routers in a POP would typically be considered part of a single group. Thus the number of groups required in a particular network becomes a function of the number of POPs and the number of backbone routers, but is independent of the number of access routers. A 100-group network therefore translates to quite a large network [5].

LSA Type	Percentage
Refresh	99.9244
Area 0 change	0.0057
Non-zero area change	0.0699

Table 2: LSA traffic breakdown for August 1, 2004

We saw more than 200,000 unique prefixes in our data. The effectiveness of the RCS shadow tables is evident by the modest rate of increase of the memory needs as the number of groups are increased. For example, storing all 203,000 prefixes for 1 group takes 175MB, while maintaining the table for 2 groups only requires an additional 21MB, because adding a group only increases the number of pointers into the global table, not the total number of unique routes maintained by the system. The total amount of memory needed for all prefixes and 100 groups is 2.2 GB, a fairly modest amount of memory by today's standards. We also show the memory requirements for networks requiring fewer prefixes.

For the BGP (only) processing considered in this subsection, we evaluate the RCS using 100 groups, all 203,000 prefixes and BGP updates only. Specifically, for these experiments the RCS used static IGP information and no OSPF related events were played back at the RCS.

Figure 9 shows BGP decision process times for 100 groups and all 203,000 prefixes for three different tests. First, the whitebox processing times are shown. The 90th percentile of the processing times for whitebox evaluation is 726 microseconds. The graph also shows the two blackbox test results, namely blackbox no queuing and blackbox realtime. As expected, the message passing adds some overhead to the processing times. The difference between the two blackbox results are due to the bursty arrival nature of the BGP updates, which produces a queuing effect on the RCS. An analysis of the BGP data show that the average number of BGP updates over 24 hours is only 6 messages per second. However, averaged over 30 second intervals, the maximum rate is much higher, going well over 100 messages per second several times during the day.

5.3 OSPF and Overall Processing

In this section, we first evaluate only the OSPF processing of RCP by considering both the performance of the OSPF Viewer and the performance of the RCS in processing OSPF-related messages. Then we evaluate the overall performance of RCP for combined BGP and OSPF related processing.

Measurement type	Area 0 change LSA	Non-zero area change LSA
Topology model	0.0089	0.0029
Intra-area SPF	0.2106	–
Inter-area SPF	0.3528	0.0559
Path cost change	0.2009	0.0053
Group change	–	0.0000
Miscellaneous	0.0084	0.0010
Total (whitebox)	0.7817	0.0653
Total (blackbox no queuing)	0.7944	0.0732
Total (blackbox realtime)	0.7957	0.1096

Table 3: Mean LSA processing time (in seconds) for the OSPF Viewer

OSPF: Recall that per LSA processing on the OSPF Viewer depends on the type of LSA. Table 2 shows the breakdown of LSA traffic into these types for August 1, 2004 data. Note that the refreshes account for 99.9% of the LSAs and require minimal processing in the OSPF Viewer; furthermore, the OSPF Viewer completely shields RCS from the refresh LSAs. For the remaining, i.e., change LSAs, Table 3 shows the whitebox, blackbox no queuing, and blackbox real-time measurements of the OSPF Viewer. The table also shows the breakdown of white-box measurements into various calculation steps.

The results in Table 3 allow us to make several important conclusions. First, and most importantly, the OSPF Viewer can process all change LSAs in a reasonable amount of time. Second, the SPF calculation and path cost change steps are the main contributors to the processing time. Third, the area 0 change LSAs take an order of magnitude more processing time than non-zero change LSAs, since area 0 changes require recomputing the path costs to every router; fortunately, the delay is still less than 0.8 seconds and, as shown in Table 2, area 0 changes are responsible for a very small portion of the change LSA traffic.

We now consider the impact of OSPF related events on the RCS processing times. Recall that OSPF events can cause the recalculation of routes by the RCS. We consider OSPF related events in isolation by playing back to the RCS only OSPF path cost changes; i.e., the RCS was pre-loaded with BGP table dumps into a realistic operational state, but no other BGP updates were played back.

Figure 10 shows RCS processing times caused by path cost changes for three different experiments with 100 router groups. Recall from Section 4.1 and Figure 6 that the sorted egress lists are used to allow the RCS to quickly find routes that are affected by a particular path cost change. The effectiveness of this scheme can be seen from Figure 10 where the 90th percentile for the whitebox processing is approximately 82 milliseconds. Figure 10 also shows the blackbox results for no queuing and realtime evaluation. As before the difference between the whitebox and blackbox no queuing results are due to the message passing overhead between the route-emulator (emulating the OSPF Viewer in this case) and the RCS. The processing times dominate relative to the message passing overhead, so these two curves are almost indistinguishable. The difference between the two blackbox evaluations suggests significant queuing effects in the RCS, where processing gets delayed because the RCS is processing earlier path cost changes, which is confirmed by an analysis of the characteristics of the path cost changes: while relatively few events occur during the day, some generate several hundred path cost changes per second. The 90th percentile of the blackbox realtime curve is 150 seconds. This result highlights the difficulty in processing internal topology changes. We discuss a more efficient way of dealing with this (the "filtered" curve in Figure 10) in Section 5.4.

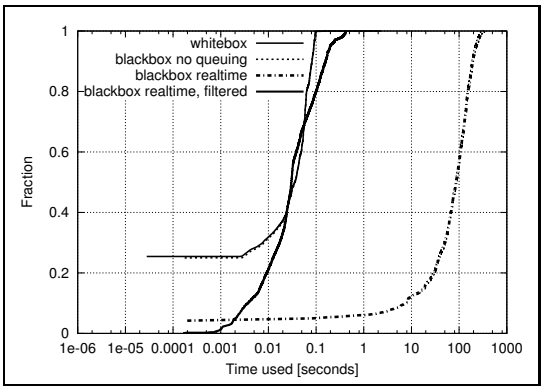


Figure 10: *Decision time, Path cost changes*: RCS route selection time for whitebox testing (instrumented RCS), blackbox testing no queuing (single path cost change sent to RCS at a time), blackbox testing real-time (path cost changes sent to RCS in real-time), blackbox testing real-time with filtered path cost changes

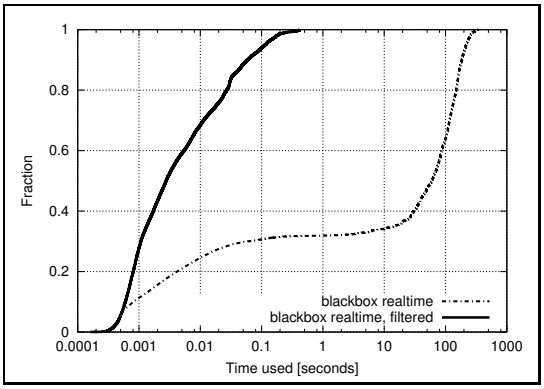


Figure 11: *Overall Processing Time, Blackbox testing BGP updates and Path cost changes combined*: All path cost changes (unfiltered) and filtered path cost changes

Overall: The above evaluation suggests that process-

ing OSPF path cost changes would dominate the overall processing time. This is indeed the case and Figure 11 shows the combined effect of playing back both BGP updates and OSPF path cost changes against the RCS. Clearly the OSPF path cost changes dominate the overall processing with the 90th percentile at 192 seconds. (The curve labeled "filtered" will be considered in the next section.)

5.4 Decoupling BGP from IGP

Although our RCP prototype handles BGP update messages very quickly, processing the internal topology changes introduces a significant challenge. The problem stems from the fact that a single event (such as a link failure) can change the IGP path costs for numerous pairs of routers, which can change the BGP route assignments for multiple routers and destination prefixes. This is fundamental to the way the BGP decision process uses the IGP path cost information to implement hot-potato routing.

The vendors of commercial routers also face challenges in processing the many BGP routing changes that can result from a single IGP event. In fact, some vendors do not execute the BGP decision process after IGP events and instead resort to performing a periodic scan of the BGP routing table to revisit the routing decision for each destination prefix. For example, some versions of commercial routers scan the BGP routing table once every 60 seconds, introducing the possibility of long inconsistencies across routers that cause forwarding loops to persist for tens of seconds [20]. The router can be configured to scan the BGP routing table more frequently, at the risk of increasing the processing load on the router.

RCP arguably faces a larger challenge from hot-potato routing changes than a conventional router, since RCP must compute BGP routes for multiple routers. Although optimizing the software would reduce the time for RCP to respond to path-cost changes, such enhancements cannot make the problem disappear entirely. Instead, we believe RCP should be used as a platform for moving beyond the artifact of hot-potato routing. In today's networks, a small IGP event can trigger a large, abrupt shift of traffic in a network [20]. We would like RCP to prevent these traffic shifts from happening, except when they are necessary to avoid congestion or delay.

To explore this direction, we performed an experiment where the RCP would not have to react to all internal IGP path cost changes, but only to those that impact the availability of the tunnel endpoint. We assume a backbone where RCP can freely direct an ingress router to any egress point that has a BGP route for the destination prefix, and can have this assignment persist across internal topology changes. This would be the case in a "BGP-free" core network, where internal routers do not have to run BGP, for example, an MPLS network or indeed any tunneled network. The edge routers in such a network still run BGP and therefore would still use IGP distances to select amongst different routes to the same destination. Some commercial router vendors accommodate this behavior by assigning an IGP weight to the tunnels and treating the tunnels as virtual IGP links. In the case of RCP, we need not necessarily treat the tunnels as IGP links, but would still need to assign some ranking to tunnels in order to facilitate the decision process.

We simulate this kind of environment by only considering OSPF path cost changes that would affect the availability of the egress points (or tunnel endpoints) but ignoring all changes that would only cause internal topology changes. The results for this experiment are shown with the *filtered* lines in Figures 10 and 11 respectively. From Figure 10, the 90th percentile for the decision time drops from 185 seconds when all path cost changes are processed to 0.059 seconds when the filtered path cost changes are used. Similarly, from Figure 11, the 90th percentile for the combined processing times drops from 192 seconds to 0.158 seconds when the filtered set is used. Not having to react to all path cost changes leads to a dramatic improvement on the processing times. Ignoring all path cost changes except those that would cause tunnel endpoints to disappear is clearly somewhat optimistic (e.g., a more sophisticated evaluation might also take traffic engineering goals into account), but it does show the benefit of this approach.

The results presented in this paper, while critically important, do not tell the whole story. From a network-wide perspective, we ultimately want to understand how long an RCP-enabled network will take to converge after a *BGP event*. Our initial results, presented in the technical report version of this paper [24], suggest that RCP convergence should be comparable to that of an iBGP route reflector hierarchy. In an iBGP topology with route reflection, convergence can actually take *longer* than with RCP in cases where routes must traverse the network multiple times before routing converges.

6 Conclusion

The networking research community has been struggling to find an effective way to redesign the Internet's routing architecture in the face of the large installed base of legacy routers and the difficulty of having a "flag day" to replace BGP. We believe that RCP provides an evolutionary path toward improving, and gradually replacing, BGP while remaining compatible with existing routers.

This paper takes an important first step by demonstrating that RCP is a viable alternative to the way BGP routes are distributed inside ASes today. RCP can emulate a full-mesh iBGP configuration while substantially reduc-

ing the overhead on the routers. By sending a customized routing decision to each router, RCP avoids the problems with forwarding loops and protocol oscillations that have plagued route-reflector configurations. RCP assigns routes consistently even when the functionality is replicated and distributed. Experiments with our initial prototype implementation show that the delays for reacting to BGP events are small enough to make RCP a viable alternative to today's iBGP architectures. We also showed the performance benefit of reducing the tight coupling between IGP path cost changes and the BGP decision process.

Acknowledgments

We would like to thank Albert Greenberg, Han Nguyen, and Brian Freeman at AT&T for suggesting the idea of an "Network Control Point for IP networks." Thanks also to Chris Chase, Brian Freeman, Albert Greenberg, Ali Iloglu, Chuck Kalmanek, John Mulligan, Han Nguyen, Arvind Ramarajan, and Samir Saad for collaborating with us on this project. We are grateful to Chen-Nee Chuah and Mythili Vutukuru, and our shepherd Ramesh Govindan, for their feedback on drafts of this paper.

7 REFERENCES

[1] N. Feamster, H. Balakrishnan, J. Rexford, A. Shaikh, and J. van der Merwe, "The case for separating routing from routers," in *Proc. ACM SIGCOMM Workshop on Future Directions in Network Architecture*, August 2004.

[2] O. Bonaventure, S. Uhlig, and B. Quoitin, "The case for more versatile BGP route reflectors." Internet Draft draft-bonaventure-bgp-route-reflectors-00.txt, July 2004.

[3] D.-F. Chang, R. Govindan, and J. Heidemann, "An empirical study of router response to large BGP routing table load," in *Proc. Internet Measurement Workshop*, November 2002.

[4] T. Bates, R. Chandra, and E. Chen, "BGP Route Reflection - An Alternative to Full Mesh IBGP." RFC 2796, April 2000.

[5] R. Dube, "A comparison of scaling techniques for BGP," *ACM Computer Communications Review*, vol. 29, July 1999.

[6] T. G. Griffin and G. Wilfong, "On the correctness of IBGP configuration," in *Proc. ACM SIGCOMM*, August 2002.

[7] A. Basu, C.-H. L. Ong, A. Rasala, F. B. Shepherd, and G. Wilfong, "Route oscillations in IBGP with route reflection," in *Proc. ACM SIGCOMM*, August 2002.

[8] D. McPherson, V. Gill, D. Walton, and A. Retana, "Border Gateway Protocol (BGP) Persistent Route Oscillation Condition." RFC 3345, August 2002.

[9] A. Shaikh and A. Greenberg, "OSPF monitoring: Architecture, design, and deployment experience," in *Proc. Networked Systems Design and Implementation*, March 2004.

[10] Ipsum Route Dynamics. `http://www.ipsumnetworks.com/route_dynamics_overview.html`.

[11] Quagga Software Routing Suite. `http://www.quagga.net`.

[12] M. Handley, O. Hudson, and E. Kohler, "XORP: An open platform for network research," in *Proc. SIGCOMM Workshop on Hot Topics in Networking*, October 2002.

[13] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click modular router," *ACM Trans. Computer Systems*, vol. 18, pp. 263–297, August 2000.

[14] R. Govindan, C. Alaettinoglu, K. Varadhan, and D. Estrin, "Route servers for inter-domain routing," *Computer Networks and ISDN Systems*, vol. 30, pp. 1157–1174, 1998.

[15] R. Govindan, "Time-space tradeoffs in route-server implementation," *Journal of Internetworking: Research and Experience*, vol. 6, June 1995.

[16] V. Jacobson, C. Alaettinoglu, and K. Poduri, "BST - BGP Scalable Transport ." NANOG27 `http://www.nanog.org/mtg-0302/ppt/van.pdf`, February 2003.

[17] N. Feamster, J. Winick, and J. Rexford, "A model of BGP routing for network engineering," in *Proc. ACM SIGMETRICS*, June 2004.

[18] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, and J. Rexford, "NetScope: Traffic engineering for IP networks," *IEEE Network Magazine*, pp. 11–19, March 2000.

[19] Y. Rekhter, T. Li, and S. Hares, "A Border Gateway Protocol 4 (BGP-4)." Internet Draft draft-ietf-idr-bgp4-26.txt, work in progress, October 2004.

[20] R. Teixeira, A. Shaikh, T. Griffin, and J. Rexford, "Dynamics of hot-potato routing in IP networks," in *Proc. ACM SIGMETRICS*, June 2004.

[21] N. Feamster and H. Balakrishnan, "Detecting BGP configuration faults with static analysis," in *Proc. Networked Systems Design and Implementation*, May 2005.

[22] J. Rexford, J. Wang, Z. Xiao, and Y. Zhang, "BGP routing stability of popular destinations," in *Proc. Internet Measurement Workshop*, November 2002.

[23] N. Spring, R. Mahajan, and D. Wetheral, "Measuring ISP topologies with RocketFuel," in *Proc. ACM SIGCOMM*, August 2002.

[24] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe, "Design and implementation of a routing control platform." `http://www.research.att.com/~kobus/rcp-nsdi-tr.pdf`, 2005.

Notes

[1] The seriousness of these inconsistencies depends on the mechanism that routers use to forward packets to a chosen egress router. If the AS uses an IGP to forward packets between ingress and egress routers, then inconsistent egress assignments along a single IGP path could result in persistent forwarding loops. On the other hand, if the AS runs a tunneling protocol (e.g., MPLS) to establish paths between ingress and egress routers, inconsistent route assignments are not likely to cause loops, assuming that the tunnels themselves are loop-free.

[2] Note that this optimization requires MED attributes to be compared across all routes in step 4 in Table 1. If MED attributes are only compared between routes with the same next-hop AS, the BGP decision process does not necessarily form a total ordering on a set of routes; consequently, the presence or absence of a non-preferred route may influence the BGP decision [17]. In this case, our optimization could cause the RCS to select a different best route than the router would in a regular BGP configuration.

[3] We filtered the BGP data so that only externally learned BGP updates were used. This represents the BGP traffic that an RCP would process when deployed.

[4] Our modular architecture would allow other BGP Engine implementations to be utilized if needed. Indeed, if required for scalability reasons, multiple BGP Engines can be deployed to "cover" a network.

[5] The per-process memory restrictions on our 32-bit platform prevented us from evaluating more groups.

Negotiation-Based Routing Between Neighboring ISPs

Ratul Mahajan David Wetherall Thomas Anderson

University of Washington

Abstract – We explore negotiation as the basis for cooperation between competing entities, for the specific case of routing between two neighboring ISPs. Interdomain routing is often driven by self-interest and based on a limited view of the internetwork, which hurts the stability and efficiency of routing. We present a negotiation framework in which adjacent ISPs share information using coarse preferences and jointly decide the paths for the traffic flows they exchange. Our framework enables pairs of ISPs to agree on routing paths based on their specific relationship, even if they have different optimization criteria. We use simulation with over sixty measured ISP topologies to evaluate our framework. We find that the quality of negotiated routing is close to that of globally optimal routing that uses complete, detailed information about both ISPs. We also find that ISPs have incentive to negotiate because *both* of them benefit compared to routing independently based on local information.

1 Introduction

A defining characteristic of the Internet (and increasingly, other planetary scale distributed systems) is that it is operated by autonomous organizations with varied interests. These organizations need to cooperate to provide a useful service, but they also compete with each other, e.g., for the same set of customers. This makes protocol design challenging, as organizations tend to hide information and make selfish policy decisions. The consequence can be both poor stability and poor efficiency.

The current interdomain routing protocol (BGP) provides an example. ISPs export little internal information and make selfish routing decisions based on their local view of the internetwork. Routing can be unstable because the actions of ISPs influence each other: in the absence of knowledge about other networks, one ISP can adversely influence another, and in the worst case cycles of influence can lead to oscillations. We are aware of one such incident that involved two large ISPs and lasted for two days [12]. Routing can be inefficient because locally sound routing decisions may be globally unsound [28, 30]. For instance, early-exit routing, in which upstream ISPs use the locally optimal exit for sending traffic to the downstream ISP, may cause the downstream ISP to carry the traffic a long way [30]. The problems with the current Internet routing architecture are also betrayed by the fact that operators are often forced to work around it by manually cooperating (as was the case in the incident above) using ad hoc mechanisms to make the routing work as desired. This manual control is neither efficient nor robust [17].

In this paper, we explore negotiation as the basis for stable and efficient routing between neighboring ISPs. This limited scenario exhibits many of the problems that occur in the more general case of interdomain routing [31], while letting us study those issues in a more tractable setting. We leave for future work the extension of our approach to cover multilateral negotiations.

With negotiation, ISPs share information in a controlled manner and jointly agree on a mutually acceptable set of paths for traffic flows they exchange. The joint agreement precludes the possibility of a cycle of influence by design. We present a practical negotiation framework, Nexit, with several properties that make it a good fit for interdomain routing. It requires ISPs to share relatively little information with each other: coarse, opaque preferences rather than transparent metrics such as latency or cost. It is flexible enough for the ISPs to reach an operating point based on their specific relationship, and it enables ISPs to optimize for their own criteria, e.g., increasing performance versus reducing cost. It also allows an ISP to ensure that it is no worse off than the default case of selfish routing with local information, so that negotiating carries no risk.

We evaluate negotiation using simulation over sixty measured ISP topologies. For both distance and bandwidth metrics, we compare negotiated routing with globally optimal routing that uses complete information to optimize the two ISPs as a single larger system. We find that the quality of negotiated routing is very close to that of the globally optimal routing. For bandwidth measures, the benefit of cooperative routing is often substantial, reducing the likelihood of overload inside either ISP. For distance measures, this benefit is small in aggregate, implying that the average "price of anarchy" [23] from a distance perspective is low in practice. The main benefit of negotiation in this setting is that it can automatically optimize a small fraction of flows with circuitous default paths. Compared to routing based on local information, both ISPs benefit with negotiation, which provides a

strong incentive to negotiate. In contrast, with global optimization, one ISP may lose to benefit the other; losing ISPs will be averse to global optimization.

Our work provides a case study of when negotiation might help to coordinate the actions of competing organizations that must cooperate to provide some service. In our case, negotiation is successful because the interests of the ISPs are not completely opposed. By cooperating, both of them benefit relative to selfish routing based solely on local information.[1] Further, we find that gains are possible only if the ISPs take a holistic view of traffic. Optimizing a single flow often means a gain for one ISP and a smaller loss for the other. Both the ISPs can gain when routing is optimized across a set of flows (as is the case for negotiation): each ISP gains for some flows and loses for others, with an overall positive gain. Nexit leverages these properties.

The rest of the paper is organized as follows. In Section 2, we provide a brief background of interdomain routing and motivate the need for better cooperation. We discuss our design considerations in Section 3 and describe our negotiation framework in Section 4. In Section 5, we empirically demonstrate the benefits of negotiation. We discuss some issues concerning deployment in Section 6, discuss related work in Section 7, and conclude in Section 8.

2 Background and Motivation

In this section, we provide a brief background of interdomain routing and give examples of problems that stem from selfish routing based on local information.

2.1 Background

For our purposes, the Internet is a collection of ISP networks or autonomous systems (ASes). We refer to inter-ISP links as *interconnections*. It is common for two large ISPs to have multiple interconnections, e.g., in different cities. ISPs use the BGP protocol to exchange reachability information – the list of ISPs along the path to the destination (known as the AS-path) – with each other to provide global connectivity. Routing information flows in the opposite direction to data flow, from downstream ISPs to upstream ISPs.

When multiple paths to a destination are available, ISPs use a combination of local policy, AS-path length and local resource constraints to select the path. The commercial relationship with the adjacent ISP is an important consideration for local policy. Typical relationships include *customer-provider*, *peers*, and *siblings*. In the first, the customer pays the provider ISP. Money is not exchanged in the other two, based on the assumption of mutual benefit for traffic exchange. Peers are often competitors that benefit from direct access to each other's customers, while siblings are friendly or related networks. Usually, ISPs prefer to send traffic through their customers, peers, and providers in that order [11]. Within these groups, paths are chosen based on their length and the amount of local resources consumed.

The original design of BGP [15] allowed only AS-path reachability information to be shared. This proved to be a serious shortcoming because ISPs want to optimize their networks, for instance, to balance load in their network, to improve the performance of the traffic they carry, or to reduce overall resource consumption. While ISPs could arbitrarily control their outgoing traffic, the inability to control incoming traffic hindered optimization. Over time, many ad hoc mechanisms have been added to address this problem.

Two such mechanisms that are commonly used today are multi-exit discriminators (MEDs) and AS-path prepending. MEDs are used between ISPs that connect in multiple locations. The downstream ISP attaches an integer to route advertisements to convey its preference for a specific destination (or destination prefix) to use a specific interconnections. If the upstream ISP chooses to honor these MEDs, it picks the best interconnection from the downstream's perspective. With AS-path prepending, the downstream ISP artificially increases the path length for traffic coming in from certain links by adding its own AS identifier multiple times in the path. Whether or not an upstream uses the increased path length in selecting paths depends on its local policies. One might think that the downstream ISP could completely determine the upstream ISP's choice by selectively advertising routes on only those interconnections it wants the upstream to use; this practice is usually prohibited by contractual agreement.

2.2 Example Problems

We now present two scenarios to illustrate the shortcomings of current interdomain routing mechanisms.

Our first example concerns the tuning of traffic exchanged between two ISPs to use resources more efficiently or to improve performance. Consider the two ISPs shown in Figure 1a, each using the closest interconnection ("early-exit") to transfer traffic to the downstream ISP as it minimizes resource usage in the upstream network. This is a common policy [30]. However, the gains of this strategy vanish when one considers traffic flowing in the reverse direction, if the other ISP also uses early-exit routing. This situation is shown in Figure 1a. Compared to a judicious choice of interconnection, as in Figure 1c, early-exit routing can lead to greater resource consumption for both ISPs and poorer overall performance because it may route traffic away from the ultimate destination. Under certain topological assumptions the cost of early exit routing can be up to three times

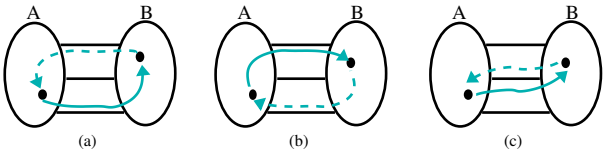


Figure 1: *Negotiation for performance tuning. (a) The default (early-exit) scenario. (b) The traffic pattern with MEDs (late-exit). (c) A mutually beneficial solution.*

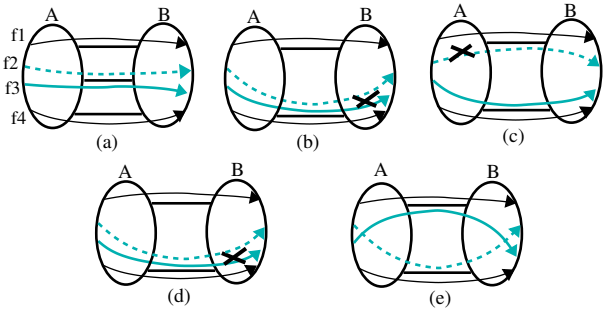


Figure 2: *Negotiation in response to failures. (a) The stable (no failure) scenario. (b) ISP-A's response to the failure of the middle interconnection congests ISP-B. (c) ISP-B's reaction of moving some traffic from the bottom interconnection to the top one congests ISP-A. (d) ISP-A reacts to its congestion by moving the traffic back to the bottom link, which again congests ISP-B. (e) A mutually acceptable solution.*

that of the optimal routing [13], though we show that it is much less in practice.

There is no straightforward way to achieve the optimized routing of Figure 1c with BGP. For instance, the use of MEDs leads to late-exit routing shown in Figure 1b. When the ISPs agree to honor each other's preferences for incoming traffic, the traffic will use the link that is closest to the destination. Done consistently, this situation is simply the reverse of early-exit.

Obtaining the routing configuration of Figure 1c requires both information sharing and coordination between ISPs. The former is not sufficient by itself as an ISP has no incentive to use the middle interconnection unless the other ISP also does the same. Coordination can convince both ISPs to give up their selfish choices.

Our second example concerns managing overload after unexpected changes in the topology or traffic such as failures or flash crowds. It is adapted from the incident mentioned in Section 1. Consider the two ISPs in Figure 2a, with traffic flowing from ISP-A to ISP-B. Assume that the middle interconnection fails, and ISP-A re-routes the affected traffic based on local conditions (Figure 2b). This overloads ISP-B, which reacts by shifting some traffic to the top interconnection, using MEDs, for instance (Figure 2c). Unfortunately, ISP-B's action overloads ISP-A, and it reacts by shifting traffic to the bottom link (Figure 2d). The result is a return to the situation of Figure 2b and continue the cycle of influence.

Figure 2e shows a solution that is acceptable to both ISPs. As before, there is no straightforward way in BGP to discover this configuration. Using MEDs, ISP-B needs to specify that the preferred entrance for $f3$ is the top interconnection and for $f2$ is the bottom one. But given a purely local view, ISP-B has no basis for preferring this configuration over $f3$ on the bottom link and $f2$ on the top one. Similarly, ISP-A has little visibility into ISP-B's network to determine the acceptable routing pattern.

3 Design Considerations

In this section, we lay out the design considerations for structuring cooperation between neighboring ISPs. Our goal is to enable the negotiating ISPs to meet their individual objectives. This implies giving them control over both their incoming *and* outgoing traffic. This control should be mutual, i.e., both upstream and downstream ISPs should be able to influence path selection. Absolute control for either the upstream or the downstream leads to problems mentioned earlier.

Our solution is based on the following key considerations that we extracted from the problem domain.

• **Limited information disclosure:** Competitors are often reluctant to disclose detailed internal information to each other. Thus, we must work with inputs that do not directly disclose unwanted information. For an ISP, this precludes disclosing information on the topology and performance of its network. This sensitivity also extends to cost information, since an ISP may not wish to tell its competitor the true cost of carrying traffic. We handle this concern by working with opaque preference classes, rather than transparent metrics such as latency or cost. An alternative approach is mechanism design, in which the best strategy for an ISP is to reveal its true cost regardless of what it knows about the other ISP's cost [8]. But this cost information can be abused outside of the solution framework, such as when an ISP adds capacity along the competitor's profitable routes [9].

• **Support for heterogeneous objectives:** Different ISPs have different goals and hence different optimization criteria. To work across such systems, cooperation mechanisms should be agnostic towards the internal objective functions used by individual entities. For instance, while ISPs with capacity constraints may aim to avoid overload, ISPs with overprovisioned networks may aim to improve performance by reducing latency and jitter. Yet others may want the best routes for their preferred customers. There are bound to be further consid-

erations of which we cannot be aware. While economic cost could be used as a unifying metric, it can be very difficult, if not impossible, for ISPs to quantify their internal considerations in terms of true cost [29]. As before, we handle this concern by working with opaque preferences. ISPs map their internal objectives to these preferences. This is relatively easier because ISPs already quantify these objectives for intradomain optimization.

A consequence of the above two considerations is that achieving social goals such as social optimality or fairness is not a pre-requisite for negotiation. Social goals are usually defined only when entities have comparable objectives. For instance, both social optimality and fairness are undefined when one ISP optimizes for latency and the other for link utilization.

• **Flexible outcomes:** As we noted in Section 2.1, different pairs of ISPs have different relationships that govern their interaction, e.g., ISPs treat customers and competitors differently. Instead of designing a mechanism that produces a deterministic output given some input, we should provide a flexible framework for ISPs to compute outcomes based on their relationship [6].

Flexibility requires that all kinds of outcomes should be possible but the most interesting space is that of "win-win" outcomes where both ISPs gain. The social optimum that treats both ISPs as a single larger system may cause one to lose compared to the situation with no negotiation. Profit-maximizing entities will not negotiate if they lose. Side payments between ISPs can alter this balance but in this paper we focus on protocols that compute win-win solutions without side payments and leave exploring the use of side payments for future work.

It is desirable that the outcomes, whatever they might be, come close to *Pareto-optimal*. An outcome is Pareto-optimal if all other outcomes are worse for at least one of the entities. Pareto-optimality rules out outcomes with obvious wastage, i.e., those that are worse for both. (The current Internet is often not Pareto-optimal as illustrated in Figure 1.) There can be multiple Pareto-optimal solutions in the system.

• **Incentive compatibility vs. efficiency:** A concern when competing entities interact is that one of them may try to manipulate the outcome in its favor by lying. Incentive compatible mechanisms, in which truth telling is provably the best strategy for all entities, guarantee interactions that are robust against manipulation. However, incentive compatibility often runs counter to efficiency. It is known that in the absence of a third party acting as a subsidizer, appraiser or arbitrator, there does not exist a mechanism that is both incentive compatible and able to implement all mutually acceptable solutions for bilateral trading [21].

Faced with the trade-off between incentive compatibility and efficiency, we favor efficiency for two reasons. First, we believe that cooperation will be the common case because parties tend to act honestly while seeking joint gains over a default contract [26]. Even today, ISPs often cooperate using ad hoc mechanisms that are not robust against manipulation. We want to compute efficient solutions when ISPs cooperate. Second, even if we were to pick incentive compatibility, it is not clear that a mechanism design approach can be used to yield flexible outcomes. Usually, for such approaches the objective of the interaction, for instance, computing least cost paths [8], is fixed by design. But we want to leave the objective up to the ISPs.

However, we will see that favoring efficiency over incentive compatibility does not necessarily imply that a cheating ISP can infinitely game the system (Sections 4.2 and 5.4).

• **Information exchange model:** Careful attention needs to be paid not only to what information is disclosed but also to how it is shared. It is virtually impossible to give ISPs mutual control over traffic in the routing information exchange model used today because routing information flows only from downstream to upstream [18]. If this information is obeyed, e.g., MEDs by contract, then the upstream loses control over outgoing traffic. If obeying this information is optional, e.g., as with AS-path prepending, whether the upstream follows it depends on its local policies. Since these policies are not known to the downstream, it cannot effectively control its incoming traffic. We use a two-way information exchange in which both the upstream and downstream ISPs provide their preference.

• **Scope of optimization:** Mutual control implies that entities have to compromise over certain issues for gain in others. What is the most effective way to arrange this mutual compromise? Economic and political negotiations tell us that better solutions are obtained when the entities negotiate over a larger set of issues [26, 3]. We find this to be true in our two-ISP scenario and encourage ISPs to keep all the traffic on the negotiating table to increase the chances of finding mutual compromises. For systems where simultaneous, mutual compromises are hard to find, compromises can be decoupled in time using "credits," a topic we leave for future work.

• **Efficient computation:** Finally, computing the result of the negotiation should be efficient in time and the number of messages. This excludes trial-and-error protocols, such as where each ISP blindly proposes to re-route a subset of the traffic at a time, in the hope that it is acceptable to the other ISP. We propose that ISPs exchange sets of preferences to efficiently discovery a mutually acceptable operating point.

4 The Nexit Framework

The goal of Nexit (short for negotiated exit) is to enable a pair of ISPs to agree upon an interconnection for each traffic flow they exchange. A *flow* is a stream of packets from a source node in one ISP to a destination node in the other ISP. There may be multiple flows between the same pair of nodes but all packets in a flow take the same path through the two networks. We discuss how ISPs establish identifiable flow signatures in Section 6. We assume that ISPs are capable of source-destination routing, i.e., flows with the same destination but different sources can be routed independently, e.g., using MPLS. By using more flexible flow definitions, Nexit can be extended to destination-based routing but for ease of exposition we focus on source-destination routing in this paper.[2]

Nexit is guided by the following observation. While improving the path of an individual flow might hurt one of the ISPs, a set of improvements will lead to a win-win solution if each improvement brings a large benefit to one ISP at a smaller cost to the other. Identifying such changes only requires the ISPs to disclose a rough measure of the cost or benefit of the change.

Conceptually, Nexit consists of two steps. First, as is the case for any negotiation, parties internally evaluate their routing choices. Second, they participate in a protocol that uses these evaluations to arrive at a mutually acceptable solution. We discuss these steps below.

1. ISP-internal evaluation of routing choices Each ISP maps flow *alternatives* to opaque preference classes based on its internal optimization criterion. An alternative corresponds to an interconnection for a flow. For example, there are three alternatives per flow for ISPs with three interconnections. Instead of using transparent metrics such as latency, Nexit works with opaque preference classes in the integral range $[-P, P]$. Internal ISP metrics are mapped to this range as described below. P is chosen to be large enough to differentiate alternatives with substantially different quality but small enough to avoid unnecessary information leakage. Opaque preferences provide a basis for negotiation between ISPs with different objectives and disclose less internal information as neither the objective nor the mapping process is revealed. But if the ISPs are interested in a social goal, they must decide in advance on the common metric and the mapping process. We consider this to be a special case for negotiation between friendly ISPs.

The mapping to preferences is done based on the default alternative for the flow, which is the alternative that the ISP reckons the flow will use in the absence of negotiation. The two ISPs need not agree on the default alternative for the flow. The ISPs map the default to preference class 0 and non-default alternatives to preferences that reflect their relative goodness.

One requirement for the mapping process is that preferences compose over addition. That is, an ISP should be happy to use two alternatives each with preference -1 if that enables it to use an alternative with preference $+3$. ISP optimization objectives of which we are aware can be mapped within this constraint. For instance, mapping per-flow objectives, such as minimizing the distance a flow traverses inside the ISP network, is straightforward as the preferences for different alternatives are independent. Network-wide objectives, such as minimizing maximum link load, can be mapped using linear program formulations [10] that optimize the sum of the individual-path preferences. Preferences for metrics that are external to an ISP network, such as those based on end-to-end path quality (gathered using measurements, for instance) can be considered mutually independent.

Preference classes are similar to BGP MEDs in terms of information disclosure, but their relative magnitude reveals some extra information. Individual ISPs can control the extent of information disclosed by using either ordinal preferences or fewer than P classes.

2. Negotiation protocol Next, the ISPs exchange their preference lists and agree on an interconnection for each flow using a protocol that proceeds in rounds. In each round, one ISP proposes an alternative and the other decides if it is acceptable. This is accomplished in several steps. The exact implementation method of each step is agreed upon contractually in advance by the ISPs.

• *Decide turn:* Decide which ISP proposes an alternative in the current round. The method we use in our experiments is that the ISPs alternate. Another is that the ISP with the lower cumulative gain (as measured using the sum of preferences for the flows negotiated so far) gets the next turn. Yet another possibility is a coin toss.

• *Propose an alternative:* The ISP whose turn it is proposes an alternative based on local and remote preferences. The method we use picks from the set that maximizes the sum of preferences of the two ISPs, breaking ties using local preferences. An alternative is to propose the best local alternative with minimal negative impact on the other ISP.

• *Accept alternative?* The other ISP decides whether to accept the proposal. This gives ISPs veto power over the proposal, which they might use if the preference for this alternative has changed since last advertised or if they perceive that the proposer is not playing by the mutually agreed rules. We always accept proposed alternatives in our experiments. Accepted flows are removed from the preference lists.

• *Reassign preferences?* Reassignment occurs when one of the ISPs wants to update its preference list. This is needed when the preferences are based on constraints such as available bandwidth that may change after some flows have been negotiated. We reassign preferences af-

Initial preference lists

	$f2_{top}$	$\mathbf{f2_{bot}}$	$f3_{top}$	$f3_{bot}$
(A, B)	(-1, 0)	**(0, 0)**	(0, 0)	(0, 0)

Reassignment after $f2_{bot}$

	$f2_{top}$	$f2_{bot}$	$\mathbf{f3_{top}}$	$f3_{bot}$
(A, B)			**(0, 1)**	(0, 0)

Figure 3: *Preference lists for the example in Figure 2. The column headings correspond to flow alternatives; the subscripts correspond to the interconnections. The tuples represent the two ISPs' preferences for that alternative. The alternative selected at each step is shown in bold.*

ter negotiating each 5% of the traffic for bandwidth experiments and do not reassign preferences for distance experiments.

• *Stop?* ISPs decide whether they want to continue negotiating over more flows. In our experiments, ISPs stop when they perceive no additional gain in continuing. We call this *early termination*. Alternately, ISPs may continue as long as their cumulative gain is positive even though it may be lower than that with early termination. We call this *full termination*. It will be preferred in interest of social welfare. The socially best outcome occurs when ISPs negotiate for all the flows, even if that means a reduction in one of the ISPs' gain.

4.1 An Example

We illustrate the working of Nexit using the second scenario of Section 2, shown in Figure 2. We simulate negotiation over the two flows, $f2$ and $f3$, impacted by the failure. Each flow has two alternatives – the top and bottom interconnections. Assume that the preference class range is [-1, 1] and the ISPs propose alternatives that maximize the total gain, breaking ties at random.

The top table in Figure 3 shows the initial preferences lists for the two ISPs. These are relative to the default of both flows traversing the bottom link. The subscripts for the flows denote the interconnection. Recall that, in that example, ISP-A is averse to $f2$ traversing the top interconnection, and ISP-B is averse to both flows coming in via the bottom interconnection. Initially, all the alternatives for ISP-A are as good as the default except $f2$ going over the top link. ISP-B is initially indifferent to all the alternatives because preference classes to flows are assigned independently of each other. ISP-B can handle either of the flows entering via the bottom link; the problem arises only when they both do. Suppose that ISP-A gets the first turn and it proposes $f2_{bot}$ by randomly picking out of the three equally good options. ISP-B accepts.

Next, the ISPs reassign preferences as shown in the bottom table: ISP-B prefers $f3_{top}$ over the default. Reassignment takes into account the expected state of the network, assuming that the first accepted choice was implemented. ISP-B takes the next turn and proposes $f3_{top}$. This alternative is accepted by ISP-A, leading to the desirable final solution shown in Figure 2e that could not be found by BGP.

Of course, Nexit may not always arrive at an exactly optimal solution. In the example, this occurs if ISP-A happens to pick $f3_{bot}$ the first time. At this point, whichever way $f2$ is routed, one of the ISPs suffers: ISP-A does not want $f2$ to use the top link and ISP-B does not want it to use the bottom link. It is possible to prevent such sub-optimality if ISPs disclose resource dependency among flows. But we opt for simplicity in the design of Nexit; we will see that for realistic scenarios, this does not lead to much efficiency loss.

4.2 Discussion

In this section, we make two observations about Nexit. First, it can be used to obtain a wide variety of outcomes. Computing exact socially optimal or Pareto optimal outcomes in our problem setting is NP-hard. The hardness for load-dependent metrics stems from the inability to split a flow across multiple paths. For load-independent metrics, computing Pareto-optimal solutions in which both ISPs do better than the default is NP-hard. This follows from a simple reduction from PARTITION; we omit this reduction due to space constraints. Nexit approximates those outcomes using its hill climbing (or greedy) structure. Socially optimal solutions are approximated when the ISPs' metrics and the mapping process are compatible (e.g., both ISPs optimize for latency and map a gain of 20ms to the same preference class), ISPs select alternatives that maximize the combined gain, and continue negotiating until all flows have been negotiated. Max-min fair solutions (that maximize the minimum gain) are approximated when the metrics and the mapping process are compatible and the ISP with lesser cumulative gain proposes alternatives, giving it a chance to catch up with the other ISP. Finally, Pareto optimal solutions are approximated when the ISPs (with possibly incompatible metrics) propose alternatives that maximize the combined gain.

Second, even though Nexit is not strictly strategy-proof in that an ISP can lie about its preferences, its structure is such that it cannot be infinitely gamed. First, a cheating ISP can never cause the truthful ISP to lose, only gain less, because the truthful ISP will not accept solutions that are worse than the default. Second, the combination of truthful ISPs that terminate negotiation when they see no more self-gain and ISPs that take turns to pick flow

alternatives may lead to premature termination of negotiation. When this occurs, it hurts the cheating ISP compared to it being truthful. Third, various modes of Nexit make cheating harder. For instance, if the alternative selection criterion is decided in advance, lying might hurt the cheater because its choices will be limited by the (incorrect) preference list that it discloses to the other ISP. In Section 5.4, we evaluate these arguments empirically. Analytically understanding the impact of cheating is an interesting avenue for future work.

5 Evaluation

In this section, we evaluate negotiated routing by comparing it with today's default routing and globally optimal routing. While the former is based on local information, the latter is based on complete information sharing and treats both ISPs as a single larger system; as such it ignores the legitimate differences in ISP interests.

We answer three high-level questions:

• *How much of the gain of globally optimal routing can be realized using negotiation, given the restrictions such as limited information sharing?* We show that negotiated routing is very close to globally optimal routing. We also show that negotiating over a large set of flows is necessary to achieve that gain.

• *Compared to the default routing, how do individual ISPs fare with globally optimal routing and with negotiation?* We show that, while the global optimal often benefits one ISP but hurts the other, negotiation always benefits both ISPs.

• *How much can a cheating ISP gain by lying about its preferences?* We show that a cheating ISP may lose compared to being truthful.

The answers to these questions depend on many aspects of ISP networks, some of which are hard to model. Our approach is to use measured data where it is available and experiment with a range of models drawn from the literature where it is not. In this way, we hope to focus on realistic rather than theoretical best- or worst-case bounds, while avoiding results that are sensitive to incidental choices in our setup.

As measured input, we use a dataset of PoP (city)-level topologies of 65 ISPs, along with geographic coordinates of PoPs and estimated inter-PoP link weights that model routing internal to an ISP [30]. This dataset is diverse in terms of ISP sizes and geographical presence.

We consider two kinds of ISP optimization criteria. The first, based on a distance metric, explores the steady-state reduction in overall network resource usage that can be achieved, implicitly assuming that the network capacity is well-matched to the traffic it carries. The second, based on a bandwidth metric, explores how negotiation can reduce the possibility of overload that might occur when the traffic is no longer well-matched to the network, e.g., due to a failure. The results from these two criteria are presented in Sections 5.1 and 5.2. Since we are interested in evaluating the potential of negotiation by comparing it to the globally optimal, which is well-defined only when ISPs use the same optimization criteria, both ISPs use the same criteria in these two sections. In Section 5.3, we evaluate the case where the two ISPs have different optimization criteria. Finally, in Section 5.4, we consider a scenario where one of the ISPs cheats by lying about its preferences.

We used Nexit as follows in our experiments. Preference class range is [-10,10]; we found that increasing the range does not lead to noticeable increase in performance. ISPs take turns to propose alternatives and pick the alternative that maximizes the gain across both of them, breaking ties using their own preferences. Proposals are always accepted as our goal is to evaluate the benefit of negotiation when ISPs cooperate fully. Preferences are not reassigned for the distance experiments and are reassigned for bandwidth experiments after negotiating each 5% of the traffic. Negotiation stops when one of the ISPs cannot gain more.

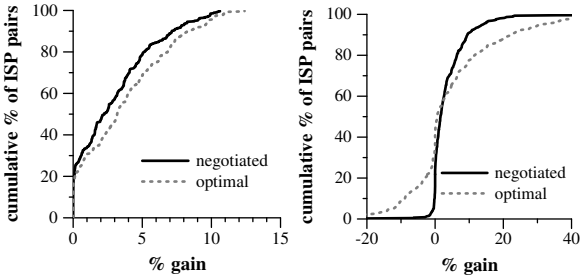
5.1 Distance and Cost

In this section, we evaluate negotiation for improving steady-state routing.

Methodology We assess the quality of steady-state routing using a metric that reflects the total resource consumption in the network. This is the sum of path lengths of all flows. There is a flow from each PoP in one ISP to each PoP in the other ISP. The length of a path is the sum of the lengths of its constituent links; we estimate link length using the geographical distance between its endpoints [22]. Our metric attempts to capture the motivation behind early-exit routing: to reduce overall network resource consumption by minimizing the distance a flow travels inside the upstream network, allowing a smaller or thinner network to support a given set of external traffic demands. Admittedly, it is a crude approximation of ISP objectives because it does not capture many factors, such as flow sizes, that ISPs might consider in practice.

For this evaluation, we use pairs of ISPs that connect at two or more locations to allow a choice of interconnections. We exclude eight ISPs whose measured topologies are a logical mesh because their geographic distance is not reflective of true distance. In all, we have 229 ISP pairs, each with traffic flows going in both directions.

We compute routing for the three methods as follows. The default routing uses the early-exit policy: the interconnection chosen by the upstream ISP for that flow is the one that is closest to the source PoP. The globally optimal routing uses the interconnection that minimizes the total distance for each flow. Negotiated routing is



(a) total gain (b) individual gain

Figure 4: *The benefit of the optimal and negotiated routing. The x-axis is the percentage reduction in the distance relative to the default routing.*

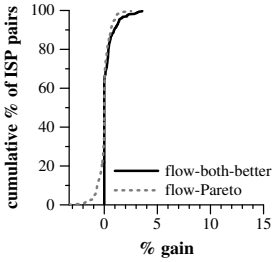


Figure 5: *The gain, relative to the default routing, of two alternate routing strategies that simply discard bad alternatives. Neither achieves nearly the potential benefit of the negotiated or optimal routing.*

computed using Nexit; ISPs use the distance of flows inside their network to map interconnection to preference classes.

Results Figure 4 shows the results of this experiment by plotting the gain of the optimal and negotiated routing relative to the default routing. The left graph plots the cumulative distribution function (CDF) of the total gain across the two ISPs. Each point corresponds to an ISP-pair. The graph shows that negotiated routing is very close to the globally optimal routing. In other words, the ISPs do not lose much by insisting that all solutions be win-win. Interestingly, however, this gain is little on average: roughly 4% for half of the ISP pairs. This suggests that the aggregate cost of early-exit routing is low, i.e., the "price of anarchy" is low for pairs of ISPs. This is well below the theoretical bound [13], probably because the topological assumptions made for computing the bound do not hold in practice. The main value of optimization in this setting is to automatically improve the performance of individual flows that suffer significantly under default routing; we consider flow-level gains shortly. We also find that, in general, ISPs with more interconnections gain more through negotiation. We omit this analysis due to space constraints.

Figure 4b plots the gain for individual ISPs in the pair. With globally optimal routing, roughly a third of the ISPs actually lose, with some losing by more than 30%. These ISPs will have little incentive to move to the globally optimal solution. In contrast, individual ISPs do not lose with negotiated routing, providing a strong incentive to negotiate.[3]

Next, we show that the gains for both ISPs depend on negotiation across a set of flows. A simpler alternative strategy would be to restrict it to pairs of flows going in the opposite direction and discard bad routing paths. We experimented with two strategies – *flow-Pareto* and *flow-both-better*. The former rejects paths that are worse than the default for both ISPs, while the latter rejects those that are worse for any one ISP. For example, in Figure 1, using the top link for $A \rightarrow B$ and the middle link for $B \rightarrow A$ is flow-Pareto, and using the middle interconnection for both directions is flow-both-better. If multiple paths satisfy the required criterion, one is picked at random. Figure 5 plots the gains from these strategies. It shows that these seemingly reasonable strategies which avoid obvious wastage at flow-level are not effective, and their cost is close to that of the default itself. We also experimented with breaking down the set of flows into several groups and negotiating within each group separately. We find that this does not provide as much benefit as negotiating over the entire set. Thus, for mutual gain to be realized, negotiation must be done across flows and ISPs must be willing to trade minor losses on some flows for significant gains on others.

We close this section with a flow-level view of negotiation. Figure 6 shows the gain for individual flows with globally optimal and negotiated routing. Some individual flows gain significantly: 7% of the flows gain over 20%, and 1% of the flows gain over 50%. We speculate that the flows that suffer heavily due to the default routing are the ones that are manually optimized by operators today. Spring *et al.* observed that a small fraction of Internet flows were routed along non-default paths between ISP-pairs [30]. Negotiation can automatically improve the performance of these flows, thus saving precious operator time. Further, the proximity of the negotiated curve to the optimal one suggests that negotiation catches almost all of the flows that need optimization.

Another interesting conclusion that can be drawn from the graph is that only a fraction of flows – roughly 20% in our experiment – need to be non-default routed to get most of the gain.

5.2 Bandwidth and Congestion

We now evaluate the benefit of negotiation in a setting where the ISPs are interested in controlling congestion and overload. Even when ISP networks are well-

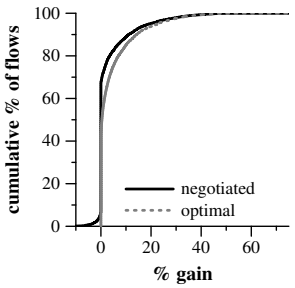


Figure 6: *A flow-level view of optimal and negotiated routing. This graph aggregates all flows across all ISP pairs.*

engineered, overload can occur during failures and sudden changes in traffic demands, as might be caused by a flash crowd [4].

Methodology We consider a scenario where an interconnection fails and simulate negotiation for the flows that are impacted by the failure; in the interest of stability, ISPs are likely to reroute only such flows. Our results may also apply to internal link failures and changes to traffic matrices.

For this experiment, we consider only ISP pairs with three or more interconnections because negotiation after a failure requires at least two working interconnections. There are 247 such ISP pairs in our dataset.

Overload is difficult to evaluate for two reasons. First, calculating bandwidth measures requires estimates of ISP link utilizations and workloads, none of which is readily available. Second, the choice of metric to represent overall ISP cost in terms of individual, congested links is less clear. For both of these, we experimented with a range of plausible models. We first describe the models used for the results presented in this paper and then list the alternate models we tried. While our results are limited to our modeling choices, we found them to be qualitatively similar for these alternate models as well, suggesting that they are not overly sensitive to our specific models.

First, we need a workload model. We assume that there is one flow from each upstream-ISP PoP to each downstream-ISP PoP; we consider only one direction of traffic at a time. To determine flow sizes we use a gravity model [19, 32], which predicts that the amount of traffic between a pair of PoPs is proportional to the product of the "weight" of the PoPs. We assume that the weight of a PoP is proportional to the population of its city. Using data from CIESIN [5], we estimate this as the number of people in a 50×50 square mile grid centered on the geographical coordinates of the city. This model leads to a skewed traffic matrix with larger cities consuming more bandwidth, both hallmarks of real Internet traffic [14, 2].

Second, to model link capacities, we assume that they are proportional to the load on the link before the failure [32], i.e., in steady-state a well-designed network tends to be roughly matched to its traffic so that links that carry more traffic tend to be of higher capacity. The traffic matrix combined with the routing within an ISP lets us compute the load on each link. But this method does not assign capacity to links in the topology that do not carry any traffic before the failure. We should not remove these links since they may be used after failures. To such links we assign a capacity that is the median of the links with non-zero load. The intuition here is that the unused links are backup links, and their capacity varies between the minimum and maximum among the links in use. Finally, to preclude our results being dominated by links that carry little traffic, we "upgrade" all links below the median to the median.

Finally, as the choice of the ISP optimization metric, we use a measure based on the intuition that ISPs prefer routing that does not significantly increase the load on links after a failure. All ISPs overprovision to some extent, so the link capacity of well-engineered networks is likely to be some small multiple of its average load. A much higher offered load after a failure implies that either the link becomes congested or it must have been significantly overprovisioned, which is expensive. Thus, our metric should penalize large increases in link load after a failure. We measure the quality of routing using maximum excess load or *MEL*, which is the maximum ratio of load after and before the failure on any link in the topology. Higher MELs are undesirable as they reflect a higher offered load on the link after the failure.

We experimented with the following alternate models. For workload, we tried identical weights for all PoPs and weights drawn from a uniform random distribution. For link capacities, we used discrete capacities by rounding them up to the nearest power of two. For assigning capacities to unused links, we used other measures such as the maximum and average load. Finally, as an alternate ISP optimization metric, we used a metric based on a linear programming formulation of optimal routing [10]. This metric minimizes the sum of link costs, where the cost is a piecewise linear function of load with increasing slope.

We reroute the impacted flows after a simulated failure using the three routing methods as follows. The default routing is early-exit over the new topology. The globally optimal is computed by solving an optimization problem that minimizes the maximum increase in link load. For computational tractability, we allow flows to be fractionally divided among interconnections; thus, the quality of this routing is an upper bound on the global optimal without fractional routing. Negotiated routing is computed using Nexit, with both ISPs using the maximum

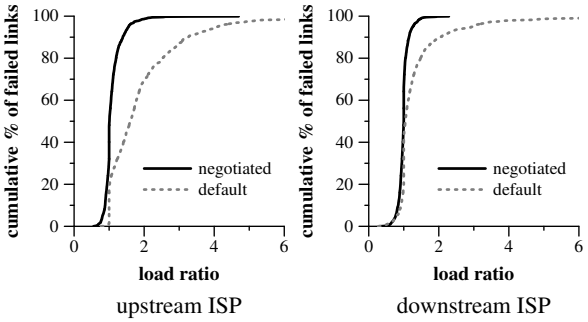


Figure 7: *The quality of negotiated routing when managing overload. The x-axis is the MEL relative to the MEL of optimal routing.*

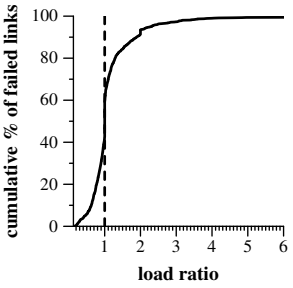


Figure 8: *The impact on downstream ISP of unilateral routing optimization by the upstream ISP. The x-axis is the ratio of the MELs for the upstream-optimized and default routing; values more than one imply that upstream-centric optimization was harmful for the downstream ISP.*

increase in link load along the path to map flows to preferences. The preferences are recomputed after each 5% of the traffic is negotiated.

Results Figure 7 shows the results of this experiment by plotting the ratio of the MEL of the default and negotiated routing to that of the optimal routing. Each data point corresponds to one hypothesized interconnection failure; so there are four distinct points for ISP pairs with four interconnections. The MEL for the default routing is often significantly larger than the optimal routing, implying that the default routing tends to overload certain links in the topology even when this overloading is avoidable. For the upstream ISP, the ratio of the two MELs is more than two for half of the cases, and more than five for 10% of the cases. Though not shown in the graph, the MEL ratios are high even when the optimal MEL is high, suggesting that overload with default routing is not limited to thin links in the network. The overload tendency is more for the upstream because many previously unused paths inside the upstream are used to send traffic from the sources to the interconnections that continue working after the failure.

The graphs also show that negotiated routing is very close to the optimal routing (since most of the MELs are one) even though the amount of information used to compute it is much less, the procedure to compute it is much simpler, and the routing itself is restrictive (compared to optimal routing which can fractionally divide a flow among interconnections).

As for distance, negotiation leads to non-default paths for only a fraction of the traffic. In our experiments, negotiation for 20% of the flows brings most of the benefit. We omit this analysis due to space constraints.

A natural question is what happens if, instead of negotiating with the downstream, the upstream unilaterally load balances outgoing traffic. It is possible that this will not hurt or may even benefit the downstream, coming close to optimal in the process. We evaluate this hypothesis by simulating the upstream ISP optimizing the routing for its own network.

Figure 8 shows the impact of upstream-centric optimization on the downstream ISP. It shows the ratio of MELs in the downstream ISP with upstream-centric optimization versus early-exit routing. The result is unpredictable: while in some cases, upstream-centric optimization helps the downstream (left end of the graph), in others the downstream ISP suffers (right end of the graph). In 10% of the cases, the MEL for upstream-centric optimization is more than twice of that for the default routing. Thus, the unilateral adjustment of routing by the upstream is undesirable because that may end up causing more congestion in the downstream. This is similar to the second example in Section 2.

5.3 Diverse Optimization Criteria

So far we have shown the quality of negotiated routing when both ISPs use the same optimization criteria, but many negotiating ISPs will use different criteria. We evaluate this case using an experiment similar to that in Section 5.2 except that the downstream ISP uses the distance metric from Section 5.1.

Figure 9 shows the results. The left graph shows how successfully the upstream ISP controls overload in its network. It plots the MEL for the default and negotiated routing relative to the MEL of optimal routing in which overload is optimized across both ISPs. The right graph shows the distance reduction in the downstream ISP relative to the default routing. Both ISPs are able to optimize for the metric of their interest. The upstream can effectively control overload and the downstream can significantly reduce the distance that the traffic traverses in its network.

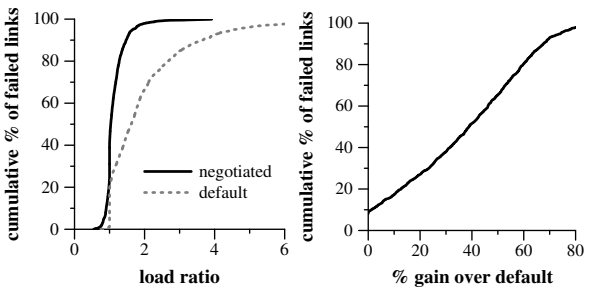


Figure 9: *Negotiation with different optimization criteria. The upstream ISP optimizes for bandwidth and the downstream ISP for distance. The x-axis of the left graph is the MEL relative to the MEL of optimal routing and that of the right graph is distance reduction relative to the default routing.*

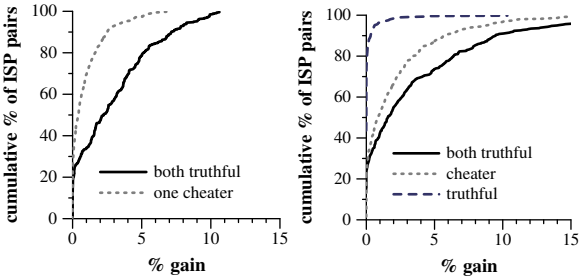


Figure 10: *The impact of cheating for the distance experiment. The x-axis is the reduction in distance compared to the default routing. (a) Reduction in distance across both ISPs. (b) Reduction in distance for individual ISPs.*

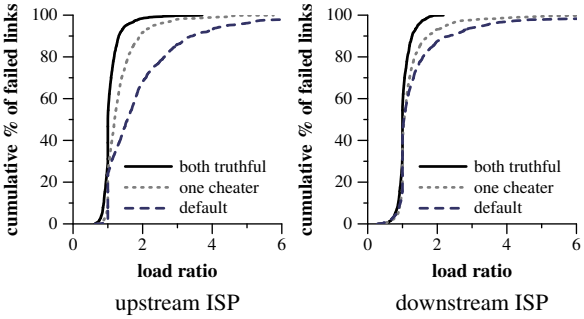


Figure 11: *The impact of cheating for the bandwidth experiment. The upstream ISP is the cheater. The x-axis is the MEL relative to that of the optimal routing.*

5.4 The Impact of Cheating

In this section, we empirically evaluate the impact of cheating on the results produced by Nexit. We do so using a cheating strategy that, on the surface at least, appears to help the cheater. Experiments with a few other strategies yielded similar results. We do not claim that this is the case for *all* possible cheating strategies.

The cheating ISP uses the following strategy in this experiment. Assume that the cheating ISP has perfect knowledge of the other ISP's preferences, which overestimates the cheater's ability because in practice some uncertainty can be introduced in this knowledge. The criterion for selecting alternatives is to maximize the sum of preferences across the two ISPs, breaking ties at random. The cheater uses the knowledge of the other ISP's preferences to inflate the preference of its best alternative for each flow just enough so that it corresponds to maximum sum. This is a better strategy than blindly maximizing preferences because as far as possible the relative ordering of the cheater's original preferences are preserved, which is useful for ensuring that better alternatives are picked first. When the other ISP's preferences are such that inflating the best alternative does not lead to maximum sum, the cheater decreases the preferences for the other alternatives accordingly.

We use the above cheating strategy for both the distance and bandwidth experiments of Sections 5.1 and 5.2. Figure 10 shows the impact of cheating for the distance experiment. The right graph shows that while cheating significantly reduces the gain of the truthful ISPs, it is unattractive as it also reduces the gain for the cheating ISPs. Figure 11 shows the impact of cheating for the bandwidth experiment with the upstream ISP acting as the cheater. As before, cheating reduces the benefit for not only the truthful, downstream ISPs but also for the cheating, upstream ISPs.

In both the experiments above, the cheating ISP loses because the negotiation terminates prematurely as the truthful ISP stops when it sees no benefit for itself. Assuming that the cheating ISP is interested in maximizing its gain, rather than minimizing the other ISP's gain, this provides a disincentive against cheating. Further, recall that even if there exist strategies by which a cheater gains, the structure of Nexit is such that an honest ISP can always protect itself by not negotiating loses.

6 Deployment Considerations

In this section, we outline how Nexit might be integrated into the current Internet. While we do not present a detailed design, we discuss several key issues concerning a practical deployment to argue for its plausibility.

Integration with ISP routing Instead of an in-band integration with BGP, we advocate an out-of-band integration with routing as shown in Figure 12. Negotiation agents use the current state of the network to map routing

alternatives to preference classes and disclose these preferences. Once the path has been negotiated, low-level BGP mechanisms such as *local-prefs* are used to implement it. This architecture is similar to RCP [7] and has three important advantages in our context. First, negotiation requires a holistic view of traffic (with ISPs losing on some flows and gaining on others) which is more cleanly accomplished with a centralized approach. Second, it avoids overloading an already fragile BGP. Third, it does not require ISPs to modify the bulk of deployed routers to benefit from negotiation.

Identifying flows for negotiation The ISPs partition the traffic they exchange into flows. Recall that a flow is a stream of packets from a node in one ISP to a node in the other. We need identifiable flow signatures because ISPs typically do not know where packets enter or leave the other ISP's network. Routing prefixes provide a basis for such a signature. Assume that the two ISPs agree on a common set of prefixes, for instance, the union of the prefixes they announce to each other through BGP. Also assume that if the prefix is aggregated, the subprefixes attach to the aggregating network at the same place.

A flow is uniquely identified using the (most specific) source and destination prefixes of its packets and an identifier that corresponds to its ingress into the upstream. When traffic that is not covered by an existing flow is observed, the upstream signals the arrival of a new flow. It informs the downstream of the two prefixes (which the downstream can also observe independently), its choice of the identifier, and the estimated flow size. To prevent information leakage, the upstream chooses different identifiers for different flows that enter at the same place.

The upstream periodically refreshes the information on active flows and flows that are inactive for a certain period are timed out.

As a practical matter, to improve scalability ISPs can decide to negotiate over only the set of long-lived and high-bandwidth (or important) flows. For this, the upstream will trigger a new flow only if its size stays above a threshold for a certain period of time. Optimizing the small fraction of high-bandwidth flows can optimize most of the traffic [31].

Input data The input data required for negotiation depend on the ISP optimization criteria but most of it can be obtained using today's technology. For instance, the network path of a given flow can be computed using the current routing state (e.g., OSPF weights or MPLS configuration). The distance of a flow through the network can be computed using the distance of individual edges. Link utilization can be obtained using SNMP probes. Information on existing flows and their sizes can be gathered using NetFlow or similar tools.

When to negotiate? While we have conceptually described negotiation as being a one-shot event between neighboring ISPs, in practice, it will be a continuous process. ISPs inform each other of their updated preferences for each flow being exchanged. These would be used to continually find routing patterns that benefit both ISPs.

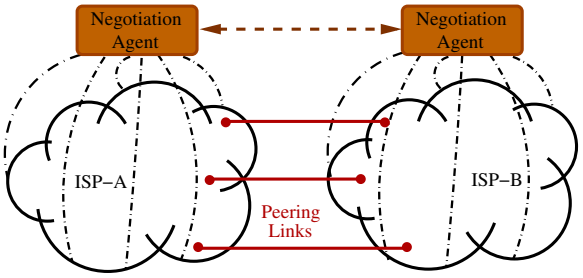


Figure 12: *Integrating negotiation with current ISP routing. Logically, the negotiation agents sit on top of the routing infrastructure. They collect data concerning the state of the network as inputs to negotiation and appropriately configure the routers to implement the negotiated solution.*

Dealing with changes Certain events, such as failures or increase in traffic quantity, require an ISP to change where or how much traffic it sends to its neighbor. There are two ways to address such events. First, the ISP informs its neighbors of the upcoming changes, negotiates with them and routes accordingly. This ensures that ISPs do not violate each other's resource constraints. However, waiting for negotiation to end before routing the flows could lead to heavy packet loss or delay if there are no alternate paths. Thus, this method is more appropriate when alternate paths exist. The second method is more suited for unplanned changes: the ISP simultaneously routes the flows and opens up a negotiation channel with the other ISP. The two ISPs then negotiate, at the end of which the flows may be rerouted. This enables faster restoration of service, with the danger that one ISP might initially overload the other. ISPs can protect other traffic from such transient overloads by lowering the priority of non-negotiated traffic entering their network.

ISPs can easily verify whether the traffic exchange complies with what was negotiated. If unilateral changes are detected (without a renegotiation request as described above), the ISP can partially or fully rollback the compromises made in return.

7 Related Work

Two works have addressed the problem of enabling neighboring domains to cooperatively manage the traffic they exchange. First, Machiraju and Katz use secure multi-party computation (SMPC) to enable ISPs to select interconnections without directly disclosing private

information [16]. In contrast to our approach, they assume that both ISPs optimize for the same objective and do not consider the possibility of an ISP losing through cooperation. In the future, we plan to explore combining SMPC with negotiation to design mechanisms that are both flexible and do not require the disclosure of internal metrics.

Second, Winick *et al.* propose a method in which before moving traffic an upstream ISP informs the downstream of changes it intends to make [31]. The downstream decides if those changes are acceptable. This method is one form of negotiation. Instead, Nexit uses preference lists to compute a mutually acceptable solution. This is useful because the solution space is exponential in the number of flows and it is hard for one ISP to find good solutions without input from the second.

Mechanisms that enable autonomous entities to cooperate have received much attention in recent years. A popular approach is distributed algorithmic mechanism design (DAMD) [9]. The two applications of this approach, of which we aware, are both *direct* mechanisms in which the entities disclose their costs [8, 24]. These costs are used to compute solutions that satisfy the desired property such as social optimality. Competitive concerns are addressed by proving that a cheating entity cannot manipulate the solution in its favor even with the knowledge of others' costs. However, this may not capture all real-world competitive concerns [9]. For instance, an ISP can use the knowledge of the competitor's cost to plan its own network in a way that undercuts the competitor's profits. We address this concern by disclosing only coarse grained, opaque preference classes. Another advantage of our approach is that the ISPs do not need to map their optimization metric to true cost, which can be difficult, if not impossible [29].

Researchers have advocated the use of money as the basis for interdomain traffic control [1, 20]. These works propose that downstreams advertise the price of carrying traffic as part of routing announcements and upstreams pay this price while sending traffic. This approach assumes that ISPs are able and willing to advertise path prices. Additionally, this approach appears to be incompatible with the charging model in the Internet, in which monetary payments flow from customer to provider ISPs irrespective of the direction of the traffic.

Our work is another piece in the research theme that examines the "price of anarchy" in the Internet. While other researchers have studied selfish routing by individual users [27, 25], we study selfish routing by ISPs. Johari and Tsitsiklis use a graphical argument to show that the latency of early-exit routing can be three times that of optimal routing [13]. Our results over real ISP topologies show that this is much lower in practice.

Finally, we draw broadly on concepts in negotiation theory [3, 21, 26], though our specific techniques are geared towards our problem domain.

8 Conclusions

In this paper, we have explored negotiation as the basis for cooperation between competing entities. Our focus has been two neighboring ISPs with multiple interconnections, which forms the base case for interdomain routing in the Internet. We presented Nexit, an inter-ISP negotiation framework in which ISPs disclose only coarse, opaque preference classes, much like BGP MEDs, to each other and jointly decide paths for the flows they exchange.

Using simulation with over sixty measured ISP topologies, for both bandwidth and distance metrics, we showed that the quality of negotiated routing is close to that of globally optimal routing which considers both ISPs to be part of one larger system. The success of negotiation stems from the fact that ISPs can trade small losses for significant gains; when applied across flows this leads to a net gain for both ISPs. While globally optimal routing can lead to both winners and losers, both ISPs benefit with negotiation, providing a strong incentive to negotiate. The benefit is often substantial for bandwidth measures, lessening the likelihood of congestion in either network. The benefit for distance is small on average, suggesting that the overall "price of anarchy" is low in practice. We also showed that because of the trading nature of negotiation, an ISP that lies can lose compared to being truthful.

In the Internet routing context, negotiation has advantages beyond the more easily quantifiable performance benefits. It can increase stability as ISPs do not inadvertently violate each other's resource constraints in a way that might set off a reactionary chain of events. It relieves operators from some of the time-consuming and error-prone tasks related to route optimization. It enables ISPs to jointly optimize traffic for profitable services such as VPNs (virtual private networks). Today, such services are limited to individual providers, and thus have limited reach. As part of ongoing effort we are working towards a prototype implementation of Nexit that will work in concert with BGP.

Our work is a first step towards designing an Internet-wide negotiation mechanism and, more broadly, understanding the trade-offs involved in the design of protocols between competing yet cooperating entities. Stability and efficiency of such systems requires that the participants have a global perspective while making local decisions. Our study shows that negotiation can be highly effective towards that goal.

9 Acknowledgements

We thank David Kempe and John Zahorjan for useful discussions and Anukool Lakhina for help with computing city population. We also thank the anonymous reviewers and our shepherd, Jon Crowcroft, for their feedback.

This work was supported in part by NSF (Grant No. ANI-0133495), Cisco Systems, and Microsoft Research.

References

[1] M. Afergan and J. Wroclawski. On the benefits and feasibility of incentive based routing infrastructure. In *ACM SIGCOMM PINS*, Sept. 2004.

[2] S. Bhattacharyya, C. Diot, J. Jetcheva, and N. Taft. PoP-level and access-link-level traffic dynamics in a Tier-1 PoP. In *ACM SIGCOMM IMW*, Nov. 2001.

[3] S. J. Brams. *Negotiation Games: Applying game theory to bargaining and arbi tration*. Routeledge, 1990.

[4] C.-N. Chuah. A Tier-1 ISP perspective: Design principles & observations of routing behavior. http://sahara.cs.berkeley.edu/jun2002-retreat/chuah_talk.pdf, 2002 June.

[5] Center for International Earth and Science Information Network. http://www.ciesin.columbia.edu.

[6] D. Clark, J. Wroclawski, K. Sollins, and R. Braden. Tussle in cyberspace: Defining tomorrow's Internet. In *ACM SIGCOMM*, Aug. 2002.

[7] N. Feamster, H. Balakrishnan, J. Rexford, A. Shaikh, and J. van der Merwe. The case for separating routing from routers. In *ACM SIGCOMM FDNA*, Aug. 2004.

[8] J. Feigenbaum, C. Papadimitriou, R. Sami, and S. Shenker. A BGP-based mechanism for lowest-cost routing. In *ACM PODC*, July 2002.

[9] J. Feigenbaum and S. Shenker. Distributed algorithmic mechanism design: Recent results and future directions. In *the 6th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, Sept. 2002.

[10] B. Fortz and M. Thorup. Internet traffic engineering by optimizing OSPF weights. In *IEEE INFOCOM*, Apr. 2000.

[11] L. Gao. On inferring autonomous system relationships in the Internet. In *IEEE Global Internet Symposium*, Nov. 2000.

[12] V. Gill. Private Communication, Nov. 2003.

[13] R. Johari and J. N. Tsitsiklis. Routing and peering in a competitive Internet. Technical Report P-2570, MIT LIDS, Jan. 2003.

[14] A. Lakhina, J. Byers, M. Crovella, and I. Matta. On the geographic location of Internet resources. *IEEE JSAC*, 2003.

[15] K. Lougheed and Y. Rekhter. A border gateway protocol (BGP). RFC 1105, IETF, June 1989.

[16] S. Machiraju and R. Katz. Reconciling cooperation with confidentiality in multi-provider distributed systems. Technical Report CSD-4-1345, UC Berkeley, Aug. 2004.

[17] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. In *ACM SIGCOMM*, Aug. 2002.

[18] R. Mahajan, D. Wetherall, and T. Anderson. Towards coordinated interdomain traffic engineering. In *HotNets-III*, Nov. 2004.

[19] A. Medina, N. Taft, K. Salamatian, S. Bhattacharyya, and C. Diot. Traffic matrix estimation: Existing techniques and new directions. In *ACM SIGCOMM*, Aug. 2002.

[20] R. Mortier and I. Pratt. Incentive based inter-domain routeing. In *Internet Charging and QoS Technology Workshop (ICQT'03)*, Sept. 2003.

[21] R. B. Myerson and M. A. Satterthwaite. Efficient mechanisms for bilateral trading. *Journal of Economic Theory*, 29(2), Apr. 1983. Cited in Brams [3].

[22] V. N. Padmanabhan and L. Subramanian. An investigation of geographic mapping techniques for Internet hosts. In *ACM SIGCOMM*, Aug. 2001.

[23] C. Papadimitriou. Algorithms, games, and the Internet. In *ACM STOC*, July 2001.

[24] G. Philips, S. Shenker, and H. Tangmunarunkit. Scaling of multicast trees: Comments on the Chuang-Sirbu scaling law. In *ACM SIGCOMM*, Aug. 1999.

[25] L. Qiu, Y. R. Yang, Y. Zhang, and S. Shenker. On selfish routing in Internet-like environments. In *ACM SIGCOMM*, Aug. 2003.

[26] H. Raiffa. *The art and science of negotiation*. Harvard University Press, 1982.

[27] T. Roughgarden and E. Tardos. How bad is selfish routing? *Journal of the ACM*, 49(2), Mar. 2002.

[28] S. Savage, A. Collins, E. Hoffman, J. Snell, and T. Anderson. The end-to-end effects of Internet path selection. In *ACM SIGCOMM*, Aug. 1999.

[29] S. Shenker, D. Clark, D. Estrin, and S. Herzog. Pricing in computer networks: Reshaping the research agenda. *ACM SIGCOMM CCR*, 26(2), Apr. 1996.

[30] N. Spring, R. Mahajan, and T. Anderson. Quantifying the causes of path inflation. In *ACM SIGCOMM*, Aug. 2003.

[31] J. Winick, S. Jamin, and J. Rexford. Traffic engineering between neighboring domains. http://www.research.att.com/~jrex/papers/interAS.pdf, July 2002.

[32] Y. Zhang, M. Roughan, N. Duffield, and A. Greenberg. Fast accurate computation of large-scale IP traffic matrices from link loads. In *ACM SIGMETRICS*, June 2003.

Notes

[1] In game theory parlance, the two-ISP situation is not zero-sum but is akin to prisoner's dilemma because both players benefit from cooperation.

[2] Empirical evaluation with destination-based routing yields results similar to those in Section 5.

[3] A subtle advantage of opaque preferences is that it makes negotiation "jealousy free" because one ISP cannot determine whether the other profits more in any meaningful terms.

Detecting BGP Configuration Faults with Static Analysis

Nick Feamster and Hari Balakrishnan
MIT Computer Science and Artificial Intelligence Laboratory
{feamster, hari}@csail.mit.edu
http://nms.csail.mit.edu/rcc/

Abstract

The Internet is composed of many independent autonomous systems (ASes) that exchange reachability information to destinations using the Border Gateway Protocol (BGP). Network operators in each AS configure BGP routers to control the routes that are learned, selected, and announced to other routers. Faults in BGP configuration can cause forwarding loops, packet loss, and unintended paths between hosts, each of which constitutes a failure of the Internet routing infrastructure.

This paper describes the design and implementation of **rcc***, the* router configuration checker*, a tool that finds faults in BGP configurations using static analysis.* **rcc** *detects faults by checking constraints that are based on a high-level correctness specification.* **rcc** *detects two broad classes of faults:* route validity *faults, where routers may learn routes that do not correspond to usable paths, and* path visibility *faults, where routers may fail to learn routes for paths that exist in the network.* **rcc** *enables network operators to test and debug configurations before deploying them in an operational network, improving on the status quo where most faults are detected only during operation.* **rcc** *has been downloaded by more than sixty-five network operators to date, some of whom have shared their configurations with us. We analyze network-wide configurations from 17 different ASes to detect a wide variety of faults and use these findings to motivate improvements to the Internet routing infrastructure.*

1 Introduction

This paper describes the design, implementation, and evaluation of ***rcc***, the router configuration checker, a tool that uses static analysis to detect faults in Border Gateway Protocol (BGP) configuration. By finding faults over a distributed set of router configurations, *rcc* enables network operators to test and debug configurations before deploying them in an operational network. This approach improves on the status quo of "stimulus-response" debugging where operators need to run configurations in an operational network before finding faults.

Network operators use router configurations to provide reachability, express routing policy (*e.g.*, transit and peering relationships [28], inbound and outbound routes [3], etc.), configure primary and backup links [17], and perform traffic engineering across multiple links [14]. Configuring a network of BGP routers is like writing a distributed program where complex feature interactions occur both within one router and across multiple routers. This complex process is exacerbated by the number of lines of code (we find that a 500-router network typically has more than a million lines of configuration), by configuration being distributed across the routers in the network, by the absence of useful high-level primitives in today's configuration languages, by the diversity in vendor-specific configuration languages, and by the number of ways in which the same high-level functionality can be expressed in a configuration language. As a result, router configurations are complex and faulty [3, 24].

Faults in BGP configuration can seriously affect end-to-end Internet connectivity, leading to lost packets, forwarding loops, and unintended paths. Configuration faults include invalid routes (including hijacked and leaked routes); contract violations [13]; unstable routes [23]; routing loops [8, 10]; and persistently oscillating routes [1, 19, 35]. Section 2 discusses the problems observed in operational networks in detail. We find that *rcc* can detect many of these configuration faults.

Detecting BGP configuration faults poses several challenges. First, defining a correctness specification for BGP is difficult: its many modes of operation and myriad tunable parameters permit a great deal of flexibility in both the design of a network and in how that design is implemented in the configuration itself. Second, this high-level correctness specification must be used to derive a set of constraints that can be tested against the actual configuration. Finally, BGP configuration is distributed—analyzing how a network configuration behaves requires both synthesizing distributed configuration fragments and representing the configuration in a form that makes it easy to test con-

straints. This paper tackles these challenges and makes the following three contributions:

First, we define two high-level aspects of correctness—*path visibility* and *route validity*—and use this specification to derive constraints that can be tested against the BGP configuration. Path visibility says that BGP will correctly propagate routes for existing, usable IP-layer paths; essentially, it states that the control path is propagating BGP routes correctly. Route validity says that, if routers attempt to send data packets via these routes, then packets will ultimately reach their intended destinations.

Second, we present the design and implementation of ***rcc***. *rcc* focuses on detecting faults that have the potential to cause *persistent* routing failures. *rcc* is not concerned with correctness during convergence (since any distributed protocol will have transient inconsistencies during convergence). *rcc*'s goal is to detect problems that may exist in the steady state, even when the protocol converges to some stable outcome.

Third, we use *rcc* to explore the extent of real-world BGP configuration faults; this paper presents the first published analysis of BGP configuration faults in real-world ISPs. We have analyzed real-world, deployed configurations from 17 different ASes and detected more than 1,000 BGP configuration faults that had previously gone undetected by operators. These faults ranged from simple "single router" faults (*e.g.*, undefined variables) to complex, *network-wide* faults involving interactions between multiple routers. To date, *rcc* has been downloaded by over 65 network operators.

Although *rcc* is actually intended to be used *before* configurations are deployed, *rcc* discovered many faults that could potentially cause failures in live, operational networks. These include: (1) faults that could have caused network partitions due to errors in how external BGP information was being propagated to routers inside an AS, (2) faults that cause invalid routes to propagate inside an AS, and (3) faults in policy expression that caused routers to advertise routes (and hence potentially forward packets) in a manner inconsistent with the AS's desired policies. Our findings indicate that configuration faults that can cause serious failures are often not immediately apparent (*i.e.*, the failure that results from a configuration fault may only be triggered by a specific failure scenario or sequence of route advertisements). If *rcc* were used before BGP configuration was deployed, we expect that it would be able to detect many immediately active faults.

Our analysis of real-world configurations suggests that most configuration faults stem from three main causes. First, the mechanisms for propagating routes within a network are overly complex. The main techniques used to propagate routes scalably within a network (*e.g.*, "route reflection with clusters") are easily misconfigured. Second, many configuration faults arise because configuration is distributed across routers: even simple policy specifications require configuration fragments on multiple routers in a network. Third, configuring policy often involves low-level mechanisms (*e.g.*, "route maps", "community lists", etc.) that should be hidden from network operators.

The rest of this paper proceeds as follows. Section 2 provides background on BGP configuration. Section 3 describes the design of *rcc*. Sections 4 and 5 discuss *rcc*'s path visibility and route validity tests. Section 6 describes implementation details. Section 7 presents configuration faults that *rcc* discovered in 17 operational networks. Section 8 addresses related work, and Section 9 concludes.

2 Background and Motivation

Today's Internet comprises over 17,000 independently operated ASes that exchange reachability information using BGP [31]. BGP distributes routes to destination prefixes via incremental updates. Each router selects one best route to a destination, announces that route to neighboring routers, and sends updates when the best route changes. Each BGP update contains several attributes. These include the *destination prefix* associated with the route; the *AS path*, the sequence of ASes that advertised the route; the *next-hop*, the IP address that the router should forward packets to in order to use the route; the *multi-exit discriminator (MED)*, which a neighboring AS can use to specify that one route should be more (or less) preferred than routes advertised at other routers in that AS; and the *community* value, which is a way of labeling a route.

BGP's configuration affects which routes are originated and propagated, how routes are modified as they propagate, which route each router selects from multiple options, and how routes propagate between routers. A single AS can have anywhere from two or three routers to many hundreds of routers. A single router's configuration can range from a few hundred lines to more than 10,000 lines. In practice, a large backbone network may have more than a thousand different policies configured across hundreds of routers.

To understand the extent to which this complex configuration is responsible for the types of failures that occur in practice, we studied the archives of the North American Network Operators Group (NANOG) mailing list, where network operators report operational problems, discuss operational issues, etc. [27]. Because the list has received about 75,000 emails over the course of ten years, we first clustered the emails by thread and pruned threads based on a list of about fifteen keywords (*e.g.*, "BGP", "issue", "loop", "problem", "outage"). We then reviewed these threads and classified each of them into one or more of the categories shown in Figure 1.

This informal study shows some clear trends. First, many routing problems are caused by configuration faults. Second, the same types of problems continually appear. Third, BGP configuration problems continually perplex even ex-

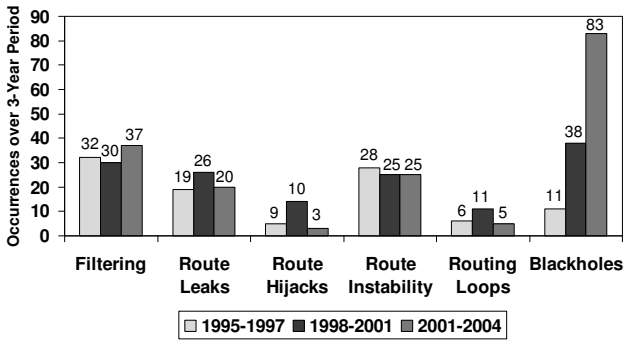


Figure 1. Number of threads discussing routing faults on the NANOG mailing list.

perienced network operators. A tool that can detect configuration faults will clearly benefit network operators.

3 *rcc* Design

rcc analyzes both single-router and network-wide properties of BGP configuration and outputs a list of configuration faults. *rcc* checks that the BGP configuration satisfies a set of constraints, which are based on a correctness specification. Figure 2 illustrates *rcc*'s high-level architecture.

We envision that *rcc* has three classes of users: those that wish to run *rcc* with no modifications, those that wish to add new constraints concerning the existing specification, and those that wish to augment the high-level specification. *rcc*'s modular design allows users to specify other constraints without changing the system internals. Some users may wish to extend the high-level specification to include other aspects of correctness (*e.g.*, safety [20]) and map those high-level specifications to constraints on the configuration.

Section 3.1 describes how we factor distributed configuration to reason about its behavior and how *rcc* generates a normalized representation of the configuration that facilitates constraint checking. To detect configuration faults, we must specify, at a high level, correct behavior for an Internet routing protocol; we outline this specification in Section 3.2. Using this high-level specification and our method for reasoning about configuration as a guide, we must then derive the actual correctness constraints that *rcc* can check against the normalized configuration. Section 3.3 explains this process.

3.1 Factoring Routing Configuration

In this section, we describe a systematic approach to analyze BGP configuration. We factor a network's configuration into the following three categories:

1. Dissemination. A router's configuration determines which other routers that router will exchange BGP routes

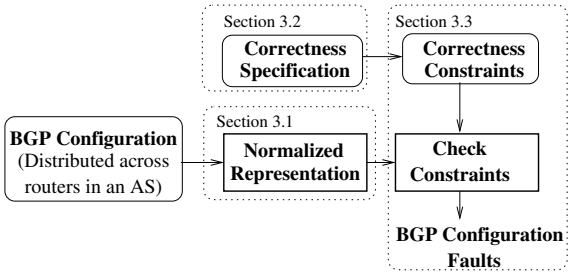


Figure 2. Overview of *rcc*.

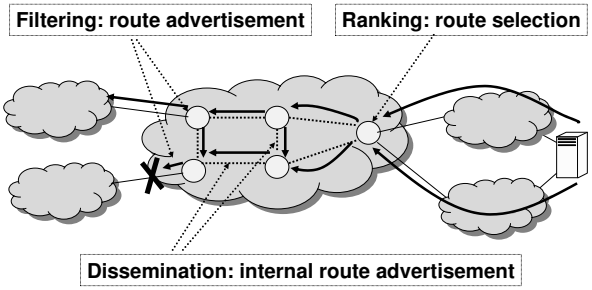


Figure 3. Factoring BGP configuration.

with. A router has two types of BGP sessions: those to routers in its own AS (internal BGP, or "iBGP") and those to routers in other ASes (external BGP, or "eBGP"). A small AS with only two or three routers may have only 10 or 20 BGP sessions, but large backbone networks typically have more than 10,000 BGP sessions, more than half of which are iBGP sessions. Dissemination primarily concerns flexibility in iBGP configuration.

The session-level BGP topology determines how BGP routes propagate through the network. In small networks, iBGP is configured as a "full mesh" (every router connects to every other router). To improve scalability, larger networks typically use *route reflectors*. A route reflector selects a single best route and announces that route to all of its "clients". Route reflectors can easily be misconfigured (we discuss iBGP misconfiguration in more detail in Section 4). Incorrect iBGP topology configuration can create persistent forwarding loops and oscillations [20].

2. Filtering. A router's configuration can prevent a certain route from being accepted on inbound or readvertised on outbound. Configuring filtering is complicated because global behavior depends on the configuration of individual routers. A router may "tag" an incoming route to control whether some other router in the AS filters the route.

3. Ranking. Any given router may learn more than one route to a destination, but must select a single best route. Configuration allows a network operator to specify which route is the most preferred route to the destination among several candidates.

Configuration also manipulates route attributes for one

Table	Description
global options	router, various global options (*e.g.*, router ID)
sessions	router, neighbor IP address, eBGP/iBGP, pointers to policy, options (*e.g.*RR client)
prefixes	router, prefix originated by this router
import/export filters	normalized representation of filter: IP range, mask range, permit or deny
import/export policies	normalized representation of policies
loopback address(es)	router, loopback IP address(es)
interfaces	router, interface IP address(es)
static routes	static routes for prefixes
Derived or External Information	
undefined references	summary of policies and filters that a BGP configuration referenced but did not define
bogon prefixes	prefixes that should always be filtered on eBGP sessions [7]

Table 1. Normalized configuration representation.

of the following reasons: (1) controlling how a router ranks candidate routes, (2) controlling the "next hop" IP address for the advertised route, and (3) labeling a route to control whether another router filters it.

rcc implements the normalized representation as a set of relational database tables. This approach allows constraints to be expressed independently of router configuration languages. As configuration languages evolve and new ones emerge, only the parser must be modified. It also facilitates testing network-wide properties, since all of the information related to the network's BGP configuration can be summarized in a handful of tables. A relational structure is natural because many sessions share common attributes (*e.g.*, all sessions to the same neighboring AS often have the same policies), and many policies have common clauses (*e.g.*, all eBGP sessions may have a filter that is defined in exactly the same way). Table 1 summarizes these tables; Section 6.1 details how *rcc* populates them.

3.2 Defining a Correctness Specification

rcc's correctness specification uses our previous work on the routing logic [10] as a starting point. *rcc* checks two aspects of correctness: *path visibility* and *route validity*. In the context of BGP, a *route* is a BGP message that advertises reachability to some destination via an associated path. A *path* is a sequence of IP hops (*i.e.*, routers) between two IP addresses. We say that a path is *usable* if it: (1) reaches the destination, and (2) conforms to the routing policies of ASes on the path.

Path visibility implies that every router learns at least one route for each destination it can reach via a usable path. Path visibility may be violated by problems with either dissemination or filtering. An example of a path visibility violation is an iBGP configuration that prevents the dissemination of BGP routes to external destinations, even though usable paths to those destinations exists.

Route validity implies that every route learned by a router describes a usable path, and that this path corresponds to the actual path taken by packets sent to the destination. Problems with dissemination or filtering can cause route validity violations. A forwarding loop is an example of a route validity violation: a router learns a route for a destination, but traffic sent on the corresponding path never reaches that destination.

rcc finds path visibility and route validity violations in BGP configuration only. To make general statements about path visibility and route validity, *rcc* assumes that the internal routing protocol (*i.e.*, interior gateway protocol, or "IGP") used to establish routes between any two routers within a AS is operating correctly. BGP requires the IGP to operate correctly because iBGP sessions may traverse multiple IGP hops and because the "next hop" for iBGP-learned routes is typically several IGP hops away.

The correctness specification that we have presented addresses *static* properties of BGP, *not* dynamic behavior (*i.e.*, its response to changing inputs, convergence time, etc.). BGP, like any distributed protocol, may experience periods of transient incorrectness in response to changing inputs. *rcc* detects faults that cause *persistent* failures. Previous work has studied sufficient conditions on the relationships between iBGP and IGP configuration that must be satisfied to guarantee that iBGP converges [20]; these constraints require parsing the IGP configuration, which *rcc* does not yet check. The correctness specifications and constraints in this paper assume that, given stable inputs, the routing protocol *eventually* converges to some steady state behavior.

Currently, *rcc* only detects faults in the BGP configuration of a *single AS* (a network operator typically does not have access to the BGP configuration from other ASes). Because an AS's BGP configuration explicitly controls both dissemination and filtering, many configuration faults, including partitions, route leaks, etc., are evident from the BGP configuration of a single AS.

3.3 Deriving Constraints and Detecting Faults

Deriving constraints on the configuration itself that guarantee that the correctness specification is satisfied is challenging. We reason about how the aspects of configuration from Section 3.1 affect each correctness property and derive appropriate constraints for each of these aspects. Specifically, Table 2 summarizes the correctness constraints that *rcc* checks, which follow from determining which aspects of configuration (from Section 3.1) affect each aspect of the correctness specification (from Section 3.2). These constraints are an attempt to map the path visibility and route validity specifications to constraints on BGP configuration that can be checked against the actual configuration.

Ideally, operators would run *rcc* to detect configuration faults *before* they are deployed. Some of *rcc*'s constraints detect faults that would most likely become active immedi-

Problem	Possible Active Fault
	Path Visibility
Dissemination Problems	
Signaling partition: - of route reflectors - within a RR "cluster" - in a "full mesh"	Router may learn a suboptimal route or none at all.
Routers with duplicate: - loopback address - cluster ID	Routers may incorrectly drop routes.
iBGP configured on one end	Routers won't exchange routes.
iBGP not to loopback	iBGP session fails when one interface fails.
	Route Validity
Filtering Problems	
transit between peers	Network carries traffic "for free".
inconsistent export to peer	Violation of contract.
inconsistent import	Possible unintentional "cold potato" routing.
eBGP session: - w/no filters - w/undef. filter - w/undef. policy	• leaked internal routes • re-advertising bogus routes • accepting bogus routes from neighbors • unintentional transit between peers
filter: - w/missing prefix	
policy: - w/undef. AS path - w/undef. community - w/undef. filter	
Dissemination Problems	
prepending with bogus AS	AS path is no longer valid.
originating unroutable dest.	Creates a blackhole.
incorrect next-hop	Other routers may be unable to reach the routes for a next-hop that is not in the IGP.
	Miscellaneous
Decision Process Problems	
nondeterministic MED age-based tiebreaking	Route selection depends on message order.

Table 2. BGP configuration problems that *rcc* detects and their potentially active faults.

ately upon deployment. For example, a router that is advertising routes with an incorrect next-hop attribute will immediately prevent other routers that use those routes from forwarding packets to those destinations. In this case, *rcc* can help the operator diagnose configuration faults and prevent them from introducing failures on the live network.

Many of the constraints in Table 2 concern faults that could remain undetected even after the configuration has been deployed because they remain masked until some sequence of messages triggers them. In these cases, *rcc* can help operators find faults that could result in a serious failure. Section 4 describes one such path visibility fault involving dissemination in iBGP in further detail. In other cases, checking constraints implies some knowledge of high-level policy (recall that a *usable* path conforms to some high-level policy). In the absence of a high-level policy specification language, *rcc* must make inferences about a network operator's intentions. Section 5 describes several route validity faults where *rcc* must make such inferences.

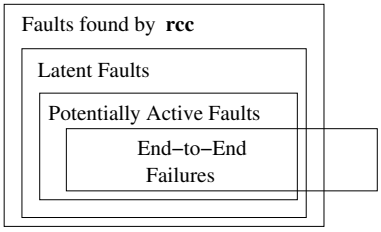


Figure 4. Relationships between faults and failures.

3.4 Completeness and Soundness

rcc's constraints are neither complete nor sound; that is, they may not find all problematic configurations, and they may complain about harmless deviations from best common practice. However, practical static analysis techniques for program analysis are typically neither complete nor sound [25]. Figure 4 shows the relationships between classes of configuration faults and the class of faults that *rcc* detects. *Latent faults* are faults that are not actively causing any problems but nonetheless violate the correctness constraints. A subset of latent faults are *potentially active* faults, for which there is at least one input sequence that is certain to trigger the fault. For example, an import policy that references an undefined filter on a BGP session to a neighboring AS is a potentially active fault, which will be triggered when that neighboring AS advertises a route that ought to have been filtered. When deployed, a potentially active fault will become *active* if the corresponding input sequence occurs. An active fault constitutes a routing failure for that AS.

Some active faults may ultimately appear as *end-to-end failures*. For example, if an AS advertises an invalid route (*e.g.*, a route for a prefix that it does not own) to a neighboring AS whose import policy references an undefined filter, then some end hosts may not be able to reach destinations within that prefix. Note that a potentially active fault may not always result in an end-to-end failure if no path between the sources and destinations traverses the routers in the faulty AS.

rcc detects a subset of latent (and hence, potentially active) faults. In addition, *rcc* may also report some false positives: faults that violate the constraints but are *benign* (*i.e.*, the violations would never cause a failure). Ideally, *rcc* would detect fewer benign faults by testing the BGP configuration against an abstract specification. Unfortunately, producing such a specification requires additional work from operators, and operators may well write incorrect specifications. One of *rcc*'s advantages is that it provides useful information about configuration faults without requiring any additional work on the part of operators.

Our previous work [10] presented three properties in addition to path visibility and route validity: information flow control (this property checks if routes "leak" in vio-

lation of policy), determinism (whether a router's preference for routes depends on the presence or absence of other routes), and safety (whether the protocol converges) [21]. This work treats information flow control as a subset of validity. *rcc* does not check for faults related to determinism and safety. Determinism cannot be checked with static analysis alone. Safety is a property that typically requires access to configurations from multiple ASes; in recent work, we have explored how to guarantee safety with access to configurations of only a single AS [11].

4 Path Visibility Faults

Recall that *path visibility* specifies that every router that has a usable path to a destination learns at least one valid route to that destination. It is an important property because it ensures that, if the network remains connected at lower layers, the routing protocol does not create any network partitions. Table 2 shows many conditions that *rcc* checks related to path visibility; in this section, we focus on iBGP configuration faults that can violate path visibility and explain how *rcc* detects these faults.

Ensuring path visibility in a "full mesh" iBGP topology is reasonably straightforward; *rcc* checks that every router in the AS has an iBGP session with every other router. If this condition is satisfied, every router in the AS will learn all eBGP-learned routes.

Because a "full mesh" iBGP topology scales poorly, operators often employ *route reflection* [2]. A subset of the routers are configured as *route reflectors*, with the configuration specifying a set of other routers as *route reflector clients*. Each route reflector readvertises its best route according to the following rules: (1) if the best route was learned from an iBGP peer, the route is readvertised to all of its route reflector clients; (2) if it was learned from a client or via an eBGP session, the route is readvertised on all iBGP sessions. A router does not readvertise iBGP-learned routes over regular iBGP sessions. If a route reflector client has multiple route reflectors, those reflectors must share all of their clients and belong to a single "cluster".

A route reflector may itself be a client of another route reflector. Any router may also have iBGP sessions with other routers. We use the set of reflector-client relationships between routers in an AS to define a graph G, where each router is a node and each session is either a directed or undirected edge: a client-reflector session is a directed edge from client to reflector, and other iBGP sessions are undirected edges. An edge exists if and only if (1) the configuration of each router endpoint specifies the loopback address of the other endpoint[1] and (2) both routers agree on session options (*e.g.*, MD5 authentication parameters). G should also not have partitions at lower layers. We say that G is *acyclic* if G has no sequence of directed and undirected edges that form a cycle. To ensure the existence of a stable path assignment, G should be acyclic.

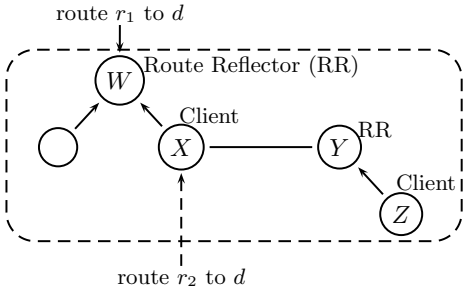


Figure 5. In this iBGP configuration, route r_2 will be distributed to all the routers in the AS, but r_1 will not. Y and Z will not learn of r_1, leading to a network partition that won't be resolved unless another route to the destination appears from elsewhere in the AS.

Even a connected directed acyclic graph of iBGP sessions can violate path visibility. For example, in Figure 5, routers Y and Z do not learn route r_1 to destination d (learned via eBGP by router W), because X will not readvertise routes learned from its iBGP session with W to other iBGP sessions. We call this path visibility fault an *iBGP signaling partition*; a path exists, but neither Y nor Z has a route for it. Note that simply adding a regular iBGP session between routers W and Y would solve the problem.

In addition to causing network partitions, iBGP signaling partitions may result in suboptimal routing. For example, in Figure 5, even if Y or Z learned a route to d via eBGP, that route might be worse than the route learned at W. In this case, Y and Z would ultimately select a suboptimal route to the destination, an event that an operator would likely fail to notice.

rcc detects iBGP signaling partitions. It determines if there is *any* combination of eBGP-learned routes such that at least one router in the AS will not learn at least one route to the destination. The following result forms the basis for a simple and efficient check.

Theorem 4.1 *Suppose that the graph defined by an AS's iBGP relationships, G, is acyclic. Then, G does not have a signaling partition if, and only if, the BGP routers that are not route reflector clients form a full mesh.*

Proof. Call the set of routers that are not reflector clients the "top layer" of G. If the top layer is not a full mesh, then there are two routers X and Y with no iBGP session between them, such that no route learned using eBGP at X will ever be disseminated to Y, since no router readvertises an iBGP-learned route.

Conversely, if the top layer is a full mesh, observe that *if* a route reflector has a route to the destination, then all its

clients have a route as well. Thus, if every router in the top layer has a route, all routers in the AS will have a route. If any router in the top layer learns a route through eBGP, then all the top layer routers will hear of the route (because the top layer is a full mesh). Alternatively, if no router at the top layer hears an eBGP-learned route, but some other router in the AS does, then that route propagates up a chain of route reflectors (each client sends it to its reflector, and the reflector sends it on all its iBGP sessions) to the top layer, from there to all the other top layer routers, and from there to the other routers in the AS. ■

rcc checks this condition by constructing the iBGP signaling graph G from the *sessions* table (Table 1). It assumes that the IGP graph is connected, then determines whether G is connected and acyclic and whether the routers at the top layer of G form a full mesh.

5 Route Validity Faults

BGP should satisfy *route validity*. Its configuration affects which routes each router accepts, selects, and readvertises. Table 2 summarizes the route validity faults that *rcc* checks. In this section, we focus on *rcc*'s approach to detecting potential policy-related problems.

The biggest challenge for checking policy-related problems is that *rcc* operates without a specification of the intended policy. Requiring operators to provide a high-level policy specification would require designing a specification language and convincing operators to use it, and it provides no guarantees that the results would be more accurate, since errors may be introduced into the specification itself. Instead, *rcc* forms *beliefs* about a network operator's intended policy in two ways: (1) assuming that intended policies conform to best common practice and (2) analyzing the configuration for common patterns and looking for deviations from those patterns. *rcc* then finds cases where the configuration appears to violate these beliefs. It is noteworthy that, even in the absence of a policy specification, this technique detects many meaningful configuration faults and generates few false positives.

5.1 Violations of Best Common Practice

Typically, a route that an AS learns from one of its "peers" should not be readvertised to another peer. Checking this condition requires determining how a route propagates through a network. Figure 6 illustrates how *rcc* performs this check. Suppose that *rcc* is analyzing the configuration from AS X and needs to determine that no routes learned from Worldcom are exported to Sprint. First, *rcc* determines all routes that X exports to Sprint, typically a set of routes that satisfy certain constraints on their attributes. For example, router A may export to Sprint only routes that are "tagged" with the label "1000". (ASes often designate such

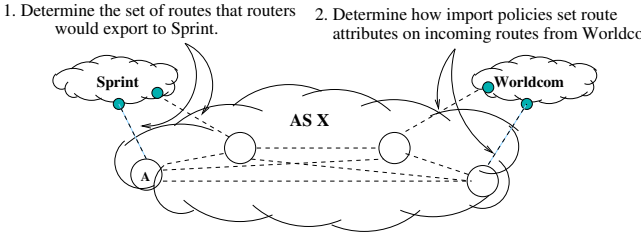


Figure 6. How *rcc* computes route propagation.

labels to signify how a route was learned.) *rcc* then checks the *import* policies for all sessions to Worldcom, ensuring that no import policy will set route attributes on any incoming route that would place it in the set of routes that would be exported to Sprint.

Additionally, an AS should advertise routes with equally good attributes to each peer at every peering point. An AS should not advertise routes with inconsistent attributes, since doing so may prevent its peer from implementing "hot potato" routing,[2] which typically violates peering agreements. Recent work has observed that this type of inconsistent route advertisement sometimes occurs in practice [13].

This violation can arise for two reasons. First, an AS may apply different export policies at different routers to the same peer. Checking for consistent export involves comparing export policies on each router that has an eBGP session with a particular peer. Static analysis is useful because it can efficiently compare policies on many different routers. In practice, this comparison is not straightforward because differences in policy definitions are difficult to detect by direct inspection of the distributed router configurations. *rcc* makes comparing export policies easy by normalizing all of the export policies for an AS, as described in Section 3.1.

Second, an iBGP signaling partition can create inconsistent export policies because routes with equally good attributes may not propagate to all peering routes. For example, consider Figure 5 again. If routers W and Z both learn routes to some destination d, then route W may learn a "better" route to d, but routers Y and Z will continue to select the less attractive route. If routers X and Y re-advertise their routes to a peer, then the routes advertised by X and Y will not be equally good. Thus, *rcc* also checks whether routers that advertise routes to the same peer are in the same iBGP signaling partition (as described in Section 4, *rcc* checks for all iBGP signaling partitions, but ones that cause inconsistent advertisement are particularly serious).

5.2 Configuration Anomalies

When the configurations for sessions at different routers to a neighboring AS are the same except at one or two routers, the deviations are likely to be mistakes. This test relies on the belief that, if an AS exchanges routes with a

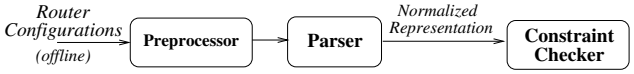


Figure 7. Overview of *rcc* implementation.

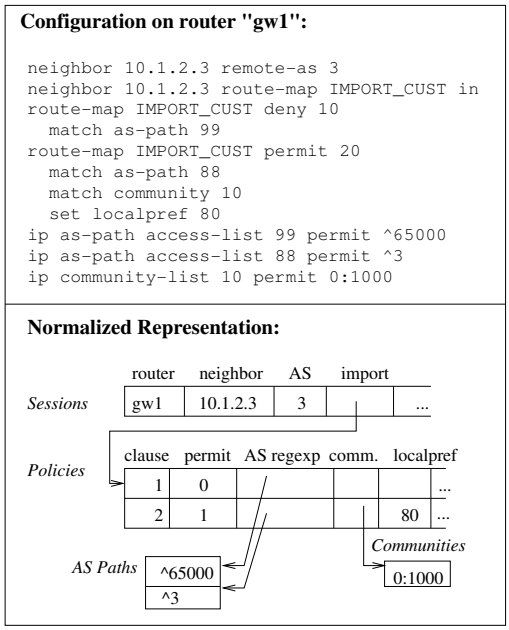


Figure 8. BGP configuration in normalized format.

neighboring AS on many sessions and most of those sessions have identical policies, then the sessions with slightly different policies may be misconfigurations. Of course, this test could result in many false positives because there are legitimate reasons for having slightly different import policies on sessions to the same neighboring AS (*e.g.*, outbound traffic engineering), but it does provide a useful sanity check.

6 Implementation

rcc is implemented in Perl and has been downloaded by over 65 network operators. The parser is roughly 60% of the code. Much of the parser's logic is dedicated to policy normalization. Figure 7 shows an overview of *rcc*, which takes as input the set of configuration files collected from routers in a single AS using a tool such as "rancid" [30]. *rcc* converts the vendor-specific BGP configuration to a vendor-independent normalized representation. It then checks this normalized format for faults based on a set of correctness constraints. *rcc*'s functionality is decomposed into three distinct modules: (1) a preprocessor, which converts configuration into a more parsable version; (2) a parser, which generates the normalized representation; and (3) a constraint checker, which checks the constraints.

6.1 Preprocessing and Parsing

The *preprocessor* adds scoping identifiers to configuration languages that do not have explicit scoping (*e.g.*, Cisco IOS) and expands macros (*e.g.*, Cisco's "peer group", "policy list", and "template" options). After the preprocessor performs some simple checks to determine whether the configuration is a Cisco-like configuration or a Juniper configuration, it launches the appropriate parser. Many configurations (*e.g.*, Avici, Procket, Zebra, Quarry) resemble Cisco configuration; the preprocessor translates these configurations so that they more closely resemble Cisco syntax.

The *parser* generates the normalized representation from the preprocessed configuration. The parser processes each router's configuration independently. It makes a single pass over each router's configuration, looking for keywords that help determine where in the configuration it is operating (*e.g.*, "`route-map`" in a Cisco configuration indicates that the parser is entering a policy declaration). The parser builds a table of normalized policies by dereferencing all filters and other references in the policy; if the reference is defined after it is referenced in the same file, the parser performs lazy evaluation. When it reaches the end of a file, the parser flags any policies references in the configuration that it was unable to resolve. The parser proceeds file-by-file (taking care to consider that definitions are scoped by each file), keeping track of normalized policies and whether they have already appeared in other configurations.

Figure 8 shows *rcc*'s normalized representation for a fragment of Cisco IOS. In *rcc*, this normalized representation is implemented as a set of mySQL database tables corresponding to the schema shown in Table 1. This Cisco configuration specifies a BGP session to a neighboring router with IP address `10.1.2.3` in AS 3. This statement is represented by a row in the ***sessions*** table. The second line of configuration specifies that the import policy (*i.e.*, "route map") for this session is defined as "`IMPORT_CUST`" elsewhere in the file; the normalized representation represents the import policy specification as a pointer into a separate table that contains the import policies themselves. A single policy, such as `IMPORT_CUST` is represented as multiple rows in the ***policies*** table. Each row represents a single clause of the policy. In this example, `IMPORT_CUST` has two clauses: the first rejects all routes whose AS path matches the regular expression number "99" (specified as "`^65000`" elsewhere in the configuration), and the second clause accepts all routes that match AS path number "88" and community number "10" and sets the "local preference" attribute on the route to a value of 80. Each of these clauses is represented as a row in the ***policies*** table; specifications for regular expressions for AS paths and communities are also stored in separate tables, as shown in Figure 8.

rcc's normalized representation does not store the names of the policies themselves (*e.g.*, "`IMPORT_CUST`", AS reg-

ular expression number "88", etc.). Rather, the normalized format only stores a description of what the route policy does (*e.g.*, "set the local preference value to 80 if the AS path matches regular expression ^3"). Two policies may be written using entirely different names, regular expression numbers, or even in different languages, but if the policies perform the same operations, *rcc* will recognize that they are in fact the same policy.

6.2 Constraint Checking

We implemented each correctness condition in Table 2 by executing SQL queries against the normalized format and analyzing the results of these queries in Perl.

rcc checks many constraints by executing simple queries against the normalized representation. Checking constraints against the normalized representation is simpler than analyzing distributed router configurations. Consider the test in Table 2 called "iBGP configured on one end"; this constraint requires that, if a router's configuration specifies an iBGP session to some IP address, then (1) that IP address should be the loopback address of some other router in the AS, and (2) that other router should be configured with an iBGP session back to the first router's loopback address. *rcc* tests this constraint as a single, simple "select" statement that "joins" the *loopbacks* and *sessions* tables. Other tests, such as checking properties of the iBGP signaling graph, require reconstructing the iBGP signaling graph using the *sessions* table.

As another example, to check that no routing policy in the AS prepends any AS number other than its own, *rcc* executes a "select" query on a join of the *sessions* and *policies* tables, which returns the ASes that each policy prepends (if any) and the routers where each policy is used. *rcc* then checks the *global* table to ensure that that for each router, the AS number configured on the router matches the ASes that any policy on that router prepends.

7 Evaluating Operational Networks with *rcc*

Our goal is to help operators move away from today's mode of stimulus-response reasoning by allowing them to check the correctness of their configurations *before* deploying them on a live network. *rcc* has helped network operators find faults in deployed configurations; we present these findings in this section. Because we used *rcc* to test configurations that were *already deployed* in live networks, we did not expect *rcc* to find many of the types of transient misconfigurations that Mahajan *et al.* found [24] (*i.e.*, those that quickly become apparent to operators when the configuration is deployed). If *rcc* were applied to BGP configurations before deployment, we expect that it could prevent more than 75% of the "origin misconfiguration" incidents and more than 90% of the "export misconfiguration" incidents described in that study.[3]

7.1 Analyzing Real-World Configurations

We used *rcc* to evaluate the configurations from 17 real-world networks, including BGP configurations from every router in 12 ASes. We made *rcc* available to operators, hoping that they would run it on their configurations and report their results.

Network operators are reluctant to share router configuration because it often encodes proprietary information. Also, many ISPs do not like researchers reporting on mistakes in their networks. (Previous efforts have enjoyed only limited success in gaining access to real-world configurations [34].) We learned that providing operators with a useful tool or service increases the likelihood of cooperation. When presented with *rcc*, many operators opted to provide us with configurations, while others ran *rcc* on their configurations and sent us the output.

rcc detected over 1,000 configuration faults. The size of these networks ranged from two routers to more than 500 routers. Many operators insisted that the details of their configurations be kept private, so we cannot report separate statistics for each network that we tested. Every network we tested had BGP configuration faults. Operators were usually unaware of the faults in their networks.

7.2 Fault Classification and Summary

Table 3 summarizes the faults that *rcc* detected. *rcc* discovered potentially serious configuration faults as well as benign ones. The fact that *rcc* discovers benign faults underscores the difficulty in specifying correct behavior. Faults have various dimensions and levels of seriousness. For example, one iBGP partition indicates that *rcc* found one case where a *network* was partitioned, but one instance of unintentional transit means that *rcc* found two *sessions* that, together, caused the AS to carry traffic in violation of high-level policy. The absolute number of faults is less important than noting that many of the faults occurred at least once.

Figure 9 shows that many faults appeared in many different ASes. We did not observe any significant correlation between network complexity and prevalence of faults, but configurations from more ASes are needed to draw any strong conclusions. The rest of this section describes the extent of the configuration faults that we found with *rcc*.

7.3 Path Visibility Faults

The path visibility faults that *rcc* detected involve iBGP signaling and fall into three categories: problems with "full mesh" and route reflector configuration, problems configuring route reflector clusters, and incomplete iBGP session configuration. Detecting these faults required access to the BGP configuration for every router in the AS.

iBGP signaling partitions. iBGP signaling partitions appeared in one of two ways: (1) the top layer of iBGP routers was not a full mesh; or (2) a route reflector cluster

Problem	Latent	Benign
Path Visibility		
Dissemination Problems		
Signaling partition:		
- of route reflectors	4	1
- within a RR "cluster"	2	0
- in a "full mesh"	2	0
Routers with duplicate:		
- loopback address	13	120
iBGP configured on one end	420	0
or not to loopback		
Route Validity		
Filtering Problems		
transit between peers	3	3
inconsistent export to peer	231	2
inconsistent import	105	12
eBGP session:		
- w/no filters	21	—
- w/undef. filter	27	—
- w/undef. policy	2	—
filter:		
- w/missing prefix	196	—
policy:		
- w/undef. AS path	31	—
- w/undef. community	12	—
- w/undef. filter	18	—
Dissemination Problems		
prepending with bogus AS	0	1
originating unroutable dest.	22	2
incorrect next-hop	0	2
Miscellaneous		
Decision Process Problems		
nondeterministic MED	43	0
age-based tiebreaking	259	0

Table 3. BGP configuration faults in 17 ASes.

had two or more route reflectors, but at least one client in the cluster did not have an iBGP session with every route reflector in the cluster. Together, these accounted for 9 iBGP signaling partitions in 5 distinct ASes, one of which was benign. While most partitions involved route reflection, we were surprised to find that even small networks had iBGP signaling partitions. In one network of only three routers, the operator had failed to configure a full mesh; he told us that he had "inadvertently removed an iBGP session". *rcc* also found two cases where routers in a cluster with multiple route reflectors did not have iBGP sessions to all route reflectors in that cluster.

rcc discovered one benign iBGP signaling partition. The network had a group of routers that did not exchange routes with the rest of the iBGP-speaking routers, but the routers that were partitioned introduced all of the routes that they learned from neighboring ASes into the IGP, rather than readvertising them via iBGP. The operator of this network told us that these routers were for voice-over-IP traffic; presumably, these routers injected all routes for this application into the IGP to achieve fast convergence after a failure or routing change. In cases such as these, BGP configuration cannot be checked in isolation from other routing protocols.

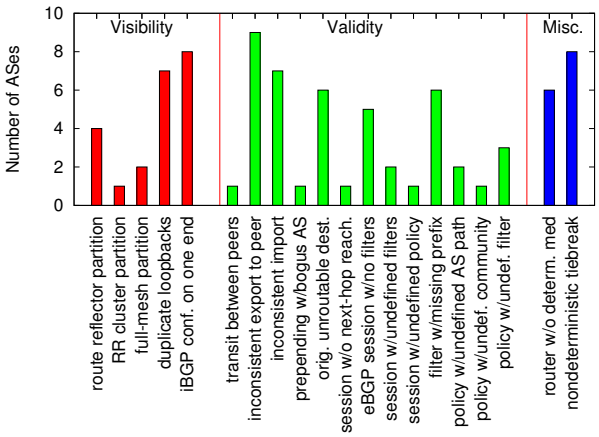


Figure 9. Number of ASes in which each type of fault occurred at least once.

Route reflector cluster problems. In an iBGP configuration with route reflection, multiple route reflectors may serve the same set of clients. This group of route reflectors and its clients is called a "cluster"; each cluster should have a unique ID, and all routers in the cluster should be assigned the same cluster ID. If a router's BGP configuration does not specify a cluster ID, then typically a router's loopback address is used as the cluster ID. If two routers have the same loopback address, then one router may discard a route learned from the other, thinking that the route is one that it had announced itself. *rcc* found 13 instances of routers in distinct clusters with duplicate loopback addresses and no assigned cluster ID.

Different physical routers in the same AS may legitimately have identical loopback addresses. For example, routers in distinct IP-layer virtual private networks may route the same IPv4 address space.

Incomplete iBGP sessions. *rcc* discovered 420 incomplete iBGP sessions (*i.e.*, a configuration statement on one router indicated the presence of an iBGP session to another router, but the other router did not have an iBGP session in the reverse direction). Many of these faults are likely benign. The most likely explanation for the large number of these is that network operators may disable sessions by removing the configuration from one end of the session without ever "cleaning up" the other end of the session.

7.4 Route Validity Faults

In this section, we discuss route validity faults. We first discuss filtering-related faults; we classify faults as latent unless a network operator explicitly told us that the fault was benign. We also describe faults concerning undefined references to policies and filters. Some of these faults, while simple to check, could have serious consequences (*e.g.*, leaked routes), if *rcc* had not caught them and they had been

activated. Finally, we present some interesting faults related to route dissemination, all of which were benign.

7.4.1 Filtering Problems

Decomposing policies across configurations on different routers can cause faults, even for simple policies such as controlling route export between peers. *rcc* discovered the following problems:

Transit between peers. *rcc* discovered three instances where routes learned from one peer or provider could be readvertised to another; typically, these faults occurred because an export policy for a session was intended to filter routes that had a certain community value, but the export policy instead referenced an undefined community.

Obsolete contractual arrangements can remain in configuration long after those arrangements expire. *rcc* discovered one AS that appeared to readvertise certain prefixes from one peer to another. Upon further investigation, we learned that the AS was actually a previous owner of one of the peers. When we notified the operator that his AS was providing transit between these two peers, he told us, "Historically, we had a relationship between them. I don't know what the status of that relationship is these days. Perhaps it is still active—at least in the configs!"

Inconsistent export to peer. We found 231 cases where an AS advertised routes that were not "equally good" at every peering point. It is hard to say whether these inconsistencies are benign without knowing the operator's intent, but roughly twenty of these inconsistencies were certainly accidental. For example, one inconsistency existed because of an undefined AS path regular expression referenced in the export policy; these types of inconsistencies have also been observed in previous measurement studies [13].

Inconsistent import policies. A recent measurement study observed that ASes often implement policies that result in late exit (or "cold potato") routing, where a router does not select the BGP route that provides the closest exit point from its own network [33].[4] *rcc* found 117 instances where an AS's import policies explicitly implemented cold potato routing, which supports this previous observation. In one network, *rcc* detected a different import policy for *every* session to each neighboring AS. In this case, the import policy was labeling routes according to the router at which the route was learned.

Inconsistent import and export policies were not always immediately apparent to us even after *rcc* detected them: the two sessions applied policies with the same name, and both policies were defined with verbatim configuration fragments. The difference resulted from the fact that the difference in policies was three levels of indirection deep. For example, one inconsistency occurred because of a difference in the definition for an AS path regular expression that the export policy referenced (which, in turn, was referenced by the session parameters).

rcc also detected filtering problems on single-router configurations:

Undefined references in policy definitions. Several large networks had router configurations that referenced undefined variables and BGP sessions that referenced undefined filters. These faults can sometimes result in unintentional transit or inconsistent export to peers or even potential invalid route advertisements. In one network, *rcc* found four routers with undefined filters that would have allowed a large ISP to accept and readvertise *any* route to the rest of the Internet (such a failure actually occurred in 1997 [32]); this potentially active fault could have been catastrophic if a customer had (unintentionally or intentionally) announced invalid routes, since ASes typically do not filter routes coming from large ISPs. This misconfiguration occurred even though the router configurations were being written with scripts; an operator had apparently made a mistake specifying inputs to the scripts. Operators can detect such faults using *rcc*.

Non-existent or inadequate filtering. Filtering can go wrong in several ways: (1) no filters are used whatsoever, (2) a filter is specified but not defined, or (3) filters are defined but are missing prefixes or otherwise out-of-date (*i.e.*, they are not current with respect to the list of private and unallocated IP address space [7]).

Every network that *rcc* analyzed had faults in filter configuration. Some of these faults would have caused an AS to readvertise *any* route learned from a neighboring AS. In one case, policy misconfiguration caused an AS to transit traffic between two of its peers. Table 3 and Figure 9 show that these faults were extremely common: *rcc* found 21 eBGP sessions in 5 distinct ASes with no filters whatsoever and 27 eBGP sessions in 2 ASes that referenced undefined filters. Every AS had partially incorrect filter configuration, and most of the smaller ASes we analyzed either had minimal or no filtering. Only a handful of the ASes we analyzed appeared to maintain rigorous, up-to-date filters for private and unallocated IP address space. These findings agree with those of our recent measurement study, which also suggests that many ASes do not perform adequate filtering [12].

The reason for inadequate filtering seems to be the lack of a process for installing and updating filters. One operator told us that he would be willing to apply more rigorous filters if he knew a good way of doing so. Another operator runs sanity checks on filters and was surprised to find that many sessions were referring to undefined filters. Even a well-defined process can go horribly wrong: one operator intended to use a feed of unallocated prefixes to automatically install filters, but instead ended up readvertising them. Because there is a set of prefixes that every AS should always filter, some prefixes should be filtered by default.

7.4.2 Dissemination Problems

We describe configuration faults involving dissemination. *rcc* found only benign faults in this case.

Unorthodox AS path prepending practices. An AS will often prepend its own AS number to the AS path on certain outbound advertisements to affect inbound traffic. However, we found one AS that prepended a neighbor's AS on *inbound advertisements* in an apparent attempt to influence *outbound traffic*.[5]

iBGP sessions with "next-hop self". We found two cases of iBGP sessions that violated common rules for setting the next-hop attribute, both of which were benign. First, *rcc* detected route reflectors that appeared to be setting the "next hop" attribute. Although this practice is not likely to create active faults, it seemed unusual, since the AS's exit routers typically set the next hop attribute, and route reflectors typically do not modify route attributes. Upon further investigation, we learned that some router vendors do not allow a route reflector to reset the next-hop attribute. Even though the configuration specified that the session would reset the next-hop attribute, the configuration statement had no effect because the software was designed to ignore it. The operator who wrote the configuration specified that the next-hop attribute be reset on these sessions to make the configuration appear more uniform. Second, routers sometimes reset the next-hop on iBGP sessions to themselves on sessions to a route monitoring server to allow the operator to distinguish which router sent each route to the monitor.

7.5 Miscellaneous Tests

Non-deterministic route selection. *rcc* discovered more than two hundred routers that were configured such that the arrival order of routes affected the outcome of the route selection process (*i.e.*, these routers had either one or both of the two configuration settings that cause nondeterminism). Although there are occasionally reasonably good reasons for introducing ordering dependencies (*e.g.*, preferring the "most stable" route; that is, the one that was advertised first), operators did not offer good reasons for why these options were disabled. In response to our pointing out this fault, one operator told us,"That's a good point, but my network isn't big enough that I've had to worry about that yet." Non-deterministic features should be disabled by default.

7.6 Higher-level Lessons

Our evaluation of real-world BGP configuration from operational networks suggests five higher-level lessons about the nature of today's configuration process. First, operational networks—even large, well-known, and well-managed ones—have faults. Even the most competent of operators find it difficult to manage BGP configuration. Moreover, iBGP is misconfigured often; in fact, in the absence of a guideline such as Theorem 4.1, it is hard for a network operator to know what properties the iBGP signaling graph should have. Second, the majority of the configuration faults that *rcc* detected resulted from the fact that an AS's configuration is distributed across its routers. A routing architecture or configuration management system that enabled an operator to configure the network from a centralized location with a high-level language would likely prevent many serious faults. Third, although operators use tools that automate some aspects of configuration, these tools are not a panacea. In fact, we found cases where the incorrect use of these tools *caused* configuration faults. Fourth, maintaining network-wide policy consistency appears to be hard; invariably, in most ASes there are routers whose configuration appears to contradict the AS's desired policy. Finally, we found that route filters are poorly maintained. Routes that should never be seen on the global Internet (*e.g.*, routes for private addresses) are rarely filtered, and the filters that are used are often misconfigured and outdated.

8 Related Work

We discuss related work in three areas: router configuration, model checking, and BGP convergence.

Router configuration. Mahajan *et al.* studied short-lived BGP misconfiguration by analyzing transient, globally visible BGP announcements from an edge network [24]. They defined a "misconfiguration" as a transient BGP announcement that was followed by a withdrawal within a small amount of time (suggesting that the operator observed and fixed the problem). They found that many misconfigurations are caused by faulty route origination and incorrect filtering. *rcc* can help operators find these faults; it can also detect faults that are difficult to quickly locate and correct. *rcc* also helps operators detect the types of misconfigurations found by Mahajan *et al.* [24] *before* deployment.

Some commercial tools analyze network configuration and highlight rudimentary errors [29]. Previous work has proposed tools that analyze intradomain routing configuration [15] and automate enterprise network configuration [6]. These tools detect router and session-level syntax errors only (*e.g.*, undefined filters), a subset of the faults that *rcc* detects. *rcc* is the first tool to check *network-wide properties* using a vendor-independent configuration representation and the first tool that applies a high-level specification of routing protocol correctness.

Many network operators use configuration management tools such as "rancid" [30], which periodically archive router configuration and provide version tracking. When a network problem coincides with the configuration change that caused it, these tools can help operators revert to an older configuration. Unfortunately, a configuration change may induce a latent or potentially active fault, and these

tools do not detect whether the configuration has these types of faults in the first place.

Model checking. Model checking has been successful in verifying the correctness of programs [18] and other network protocols [4, 22, 26]. Unfortunately, model checking is not appropriate for verifying BGP configuration because it depends heavily on exhausting the state-space within an appropriately-defined environment [25]. The behavior of an AS's BGP configuration depends on routes that arrive from other ASes, some of which, such as backup paths, cannot be known in advance [9].

Analysis of BGP safety and stability. Previous work has noted that BGP may not converge to a stable path assignment and stated sufficient conditions to guarantee that BGP will arrive at such an assignment [19, 21, 35]. This property is called *safety*. Gao and Rexford state sufficient conditions for safety in eBGP and observe that typical policy configurations satisfy these conditions [16]. (Griffin *et al.* note that analogous sufficient conditions apply to iBGP with route reflection [20].) In both cases, the sufficient conditions also require global knowledge of either rankings or the AS-level topology. *rcc* tests constraints that must hold on the configuration of a *single* AS. Our recent work derives necessary conditions that the configuration of each AS must satisfy to guarantee safety [11].

9 Discussion and Conclusion

In recent years, much work has been done to understand BGP's behavior, and much has been written about the wide range of problems it has. Some argue that BGP has outlived its purpose and should be replaced; others argue that faults arise because today's configuration languages are not well-designed. We believe that our evaluation of faults in today's BGP configuration provides a better understanding of the types of errors that appear in today's BGP configuration and the problems in today's configuration languages. Our findings should help inform the design of wide-area routing systems in the future.

Despite the fact that BGP is almost 10 years old, operators continually make the same mistakes as they did during BGP's infancy, and, regrettably, our understanding of what it means for BGP to behave "correctly" is still rudimentary. This paper takes a step towards improving this state of affairs by making the following contributions:

- We define a high-level correctness specification for BGP and map that specification to conditions that can be tested with static analysis.
- We use this specification to design and implement *rcc*, a static analysis tool that detects faults by analyzing the BGP configuration across a single AS. With *rcc*, network operators can find many faults *before* deploying configurations in an operational network. *rcc* has been downloaded by over 65 network operators.
- We use *rcc* to explore the extent of real-world BGP misconfigurations. We have analyzed real-world, deployed configurations from 17 different ASes and detected more than 1,000 BGP configuration faults that had previously gone undetected by operators.

In light of our findings, we suggest two ways to make interdomain routing less prone to configuration faults. First, protocol improvements, particularly in intra-AS route dissemination, could avert many BGP configuration faults. The current approach to scaling iBGP should be replaced. Route reflection serves a single, relatively simple purpose, but it is the source of many faults, many of which cannot be checked with static analysis of BGP configuration alone [20]. The protocol that disseminates BGP routes within an AS should enforce path visibility and route validity; the Routing Control Platform [5] offers one possible solution.

Second, BGP should be configured with a centralized, higher-level specification language. Today's BGP configuration languages enable an operator to specify router-level *mechanisms* that implement high-level policy, but the distributed, low-level nature of the configuration languages introduces complexity, obscurity, and opportunities for misconfiguration rather than design flexibility or expressiveness. For example, *rcc* detects many faults in implementation of some high-level policies in low-level configuration; these faults arise because there are many ways to implement the same high-level policy, and the low-level configuration is unintuitive. Ideally, a network operator would never touch low-level mechanisms (*e.g.*, the community attribute) in the common case. Rather than configuring routers with a low-level language, an operator should configure the *network* using a language that directly reflects high-level policies.

Acknowledgements

This work would not have been possible without the support and patience of many in the network operations community. We are grateful to Randy Bush, Jennifer Rexford, and Guy Tal for inspiration, suggestions, and feedback. We thank David Andersen, Tom Barron, Rob Beverly, Jay Borkenhagen, Grover Browning, Andres Gasson, Michael Hallgren, John Heasley, Jaroslaw Kowalczyk, Arnaud Le Taillanter, Simon Leinen, Ratul Mahajan, Hank Nussbacher, Scott Poretsky, Jeff Schiller, Nicolas Strina, and Matt Zekauskas for their help. We also thank Eddie Kohler (this paper's "shepherd"), Mike Walfish, and the anonymous NSDI reviewers for thoughtful feedback that improved this paper. This work was supported by a Cisco URP grant and by the NSF under Cooperative Agreement ANI-0225660. Nick Feamster is partially supported by an NSF Graduate Research Fellowship.

References

[1] BASU, A., ET AL. Route oscillations in IBGP with route reflection. In *Proc. ACM SIGCOMM* (Pittsburgh, PA, Aug. 2002).

[2] BATES, T., CHANDRA, R., AND CHEN, E. *BGP Route Reflection - An Alternative to Full Mesh IBGP*. Internet Engineering Task Force, Apr. 2000. RFC 2796.

[3] BEIJNUM, I. V. *BGP*. O'Reilly and Associates, Sept. 2002.

[4] BHARGAVAN, K., OBRADOVIC, D., AND GUNTER, C. A. Formal verification of standards for distance vector routing protocols. *Journal of the ACM 49*, 4 (July 2002), 538–576.

[5] CAESAR, M., FEAMSTER, N., REXFORD, J., SHAIKH, A., AND VAN DER MERWE, K. Design and Implementation of a Routing Control Platform. In *Proc. 2nd Symposium on Networked Systems Design and Implementation* (Boston, MA, May 2005).

[6] CALDWELL, D., GILBERT, A., GOTTLIEB, J., GREENBERG, A., HJALMTYSSON, G., AND REXCORD, J. The cutting EDGE of IP router configuration. In *Proc. 2nd ACM Workshop on Hot Topics in Networks (Hotnets-II)* (Cambridge, MA, Nov. 2003).

[7] Team Cymru bogon route server project. `http://www.cymru.com/BGP/bogon-rs.html`.

[8] DUBE, R. A comparison of scaling techniques for BGP. *ACM Computer Communications Review 29*, 3 (July 1999), 44–46.

[9] FEAMSTER, N. Practical verification techniques for wide-area routing. In *Proc. 2nd ACM Workshop on Hot Topics in Networks (Hotnets-II)* (Cambridge, MA, Nov. 2003).

[10] FEAMSTER, N., AND BALAKRISHNAN, H. Towards a logic for wide-area Internet routing. In *ACM SIGCOMM Workshop on Future Directions in Network Architecture* (Karlsruhe, Germany, Aug. 2003).

[11] FEAMSTER, N., JOHARI, R., AND BALAKRISHNAN, H. Stable policy routing with provider independence. Tech. Rep. MIT-LCS-TR-981, Massachusetts Institute of Technology, Feb. 2005.

[12] FEAMSTER, N., JUNG, J., AND BALAKRISHNAN, H. An empirical study of "bogon" route advertisements. *ACM Computer Communications Review* (Nov. 2004).

[13] FEAMSTER, N., MAO, Z. M., AND REXFORD, J. BorderGuard: Detecting cold potatoes from peers. In *Proc. ACM SIGCOMM Internet Measurement Conference* (Taormina, Sicily, Italy, Oct. 2004).

[14] FEAMSTER, N., WINICK, J., AND REXFORD, J. A model of BGP routing for network engineering. In *Proc. ACM SIGMETRICS* (New York, NY, June 2004).

[15] FELDMANN, A., AND REXFORD, J. IP network configuration for intradomain traffic engineering. *IEEE Network* (Sept. 2001).

[16] GAO, L. On inferring automonous system relationships in the Internet. *IEEE/ACM Transactions on Networking 9*, 6 (Dec. 2001), 733–745.

[17] GAO, L., GRIFFIN, T. G., AND REXFORD, J. Inherently safe backup routing with BGP. In *Proc. IEEE INFOCOM* (Anchorage, AK, Apr. 2001).

[18] GODEFROID, P. Model Checking for Programming Languages using VeriSoft. In *Proc. ACM Symposium on Principles of Programming Languages* (1997).

[19] GRIFFIN, T., AND WILFONG, G. An analysis of BGP convergence properties. In *Proc. ACM SIGCOMM* (Cambridge, MA, Sept. 1999).

[20] GRIFFIN, T., AND WILFONG, G. On the correctness of IBGP configuration. In *Proc. ACM SIGCOMM* (Pittsburgh, PA, Aug. 2002).

[21] GRIFFIN, T. G., SHEPHERD, F. B., , AND WILFONG, G. The stable paths problem and interdomain routing. *IEEE/ACM Transactions on Networking 10*, 1 (2002), 232–243.

[22] HAJEK, J. Automatically verified data transfer protocols. In *Proc. ICCC* (1978), pp. 749–756.

[23] LABOVITZ, C., AHUJA, A., BOSE, A., AND JAHANIAN, F. Delayed Internet Routing Convergence. *IEEE/ACM Transactions on Networking 9*, 3 (June 2001), 293–306.

[24] MAHAJAN, R., WETHERALL, D., AND ANDERSON, T. Understanding BGP misconfiguration. In *Proc. ACM SIGCOMM* (Pittsburgh, PA, Aug. 2002), pp. 3–17.

[25] MUSUVATHI, M., AND ENGLER, D. Some lessons from using static analysis and software model checking for bug finding. In *Workshop on Software Model Checking* (Boulder, CO, July 2003).

[26] MUSUVATHI, M., AND ENGLER, D. A framework for model checking network protocols. In *Proc. First Symposium on Networked Systems Design and Implementation (NSDI)* (San Francisco, CA, Mar. 2004).

[27] The North American Network Operators' Group mailing list archive. `http://www.cctec.com/maillists/nanog/`.

[28] NORTON, W. Internet service providers and peering. `http://www.equinix.com/press/whtppr.htm`.

[29] Opnet NetDoctor. `http://opnet.com/products/modules/netdoctor.htm`.

[30] Really Awesome New Cisco ConfIg Differ (RANCID). `http://www.shrubbery.net/rancid/`, 2004.

[31] REKHTER, Y., AND LI, T. *A Border Gateway Protocol 4 (BGP-4)*. Internet Engineering Task Force, Mar. 1995. RFC 1771.

[32] Router Glitch Cuts Net Access. http://news.com.com/2100-1033-279235.html, Apr. 1997.

[33] SPRING, N., MAHAJAN, R., AND ANDERSON, T. Quantifying the causes of path inflation. In *Proc. ACM SIGCOMM* (Karlsruhe, Germany, Aug. 2003).

[34] BGP config donation. `http://www.cs.washington.edu/research/networking/policy-inference/donation.html`.

[35] VARADHAN, K., GOVINDAN, R., AND ESTRIN, D. Persistent route oscillations in inter-domain routing. *Computer Networks 32*, 1 (2000), 1–16.

Notes

[1] If a router establishes an iBGP session with a router's *loopback* address, then the iBGP session will remain active as long as that router is reachable via *any* IGP path between the two routers. If a router establishes an iBGP session with an interface address of another router, however, the iBGP session will go down if that interface fails, even if an IGP path exists between those routers.

[2] If ASes 1 and 2 are peers, then the export policies of the routers in AS 1 should export routes to AS 2 that have equal AS path length and MED values. If not, router *X* could be forced to send traffic to AS 1 via router *Y* ("cold potato" routing)

[3] *rcc* detects the following classes of misconfiguration described by Mahajan *et al.*: reliance on upstream filtering, old configuration, community, forgotten filter, prefix-based config, bad ACL or route map, and typo.

[4] Inconsistent import policy technically concerns how configuration affects *ranking*, and it is more often intentional than not. Nevertheless, this test occasionally highlights anomalies that operators are interested in correcting, and it serves as a useful sanity check when looking for other types of anomalies (such as dynamically detecting inconsistent route advertisements from a neighboring AS [13]).

[5] One network operator also mentioned that ASes sometimes prepend the AS number of a network that they want to prevent from seeing a certain route (*i.e.*, by making that AS discard the route due to loop detection), effectively "poisoning" the route. We did not witness this poisoning in any of the configurations we analyzed.

IP Fault Localization Via Risk Modeling

Ramana Rao Kompella[†], Jennifer Yates[‡], Albert Greenberg[‡], Alex C. Snoeren[†]

[†]*University of California, San Diego*
{*ramana,snoeren*}*@cs.ucsd.edu*

[‡]*AT&T Labs – Research*
{*jyates,albert*}*@research.att.com*

Abstract

Automated, rapid, and effective fault management is a central goal of large operational IP networks. Today's networks suffer from a wide and volatile set of failure modes, where the underlying fault proves difficult to detect and localize, thereby delaying repair. One of the main challenges stems from operational reality: IP routing and the underlying optical fiber plant are typically described by disparate data models and housed in distinct network management systems. We introduce a fault-localization methodology based on the use of risk models and an associated troubleshooting system, SCORE (Spatial Correlation Engine), which automatically identifies likely root causes across layers. In particular, we apply SCORE to the problem of localizing link failures in IP and optical networks. In experiments conducted on a tier-1 ISP backbone, SCORE proved remarkably effective at localizing optical link failures using only IP-layer event logs. Moreover, SCORE was often able to automatically uncover inconsistencies in the databases that maintain the critical associations between the IP and optical networks.

1 Introduction

Operational IP networks are intrinsically exposed to a wide variety of faults and impairments. These networks are large, geographically distributed, constantly evolving, with complex hardware and software artifacts. A typical tier-1 network consists of about 1,000 routers from different vendors, with different features, and acting in different roles in the network architecture. Such a network is supported by access and core transport networks, which typically involve at least two orders of magnitude more network elements (optical amplifiers, Dense Wavelength Division Multiplexing (DWDM) systems, ATM/MPLS/Ethernet switches, and so forth). These network elements and associated telemetry generate a large number of management events relating to performance and potential failure conditions. The essential problem of IP fault management is to monitor the event stream to detect, localize, mitigate and ultimately correct any condition that degrades network behavior.

Unfortunately, operational IP networks today lack intrinsic robustness; serious faults and outages are not infrequent. While existing fault management systems (e.g., [12, 21, 8]) provide great value in automating routine fault management, serious problems can fly "under the radar," or, once detected, cannot be rapidly localized and diagnosed. To appreciate why this is so, it may help to imagine a network operator faced with the task of IP fault management. After much effort, network hardware has been designed and implemented, the protocols controlling the network have been designed (often in compliance with published standards), and the associated software implemented. In accord with the network architecture, the network elements have been deployed, connected, and configured. Yet, all these complex endeavors are carried out by multiple teams at rapid pace, involving a large and distributed software component, thus producing operational artifacts far richer in behavior than can ever be approximated in a lab. Errors *will* be introduced at each stage of network definition and go undetected despite best practices in design, implementation, and testing. External factors, including bugs of all types (memory leaks, inadequate performance separation between processes, etc.) in router software and environmental factors such as DoS attacks and BGP-related traffic events originating in peer networks significantly raise the level of difficulty. It is the task of IP fault management to cope with the result, continually learning and dealing with new failure modes in the field.

In this paper, we introduce a risk modeling methodology that allows for faster, more accurate automatic localization of IP faults to support both real-time and offline analysis. By design, we:

- split our solution into generic algorithmic components (Section 4.2) and problem domain specific components (Section 4.5), and
- create risk models that reflect fundamental architectural elements of the problem domain, but not necessarily implementation details.

As a result, our system is robust to churn in operational networks and is more likely to be extensible to additional system components.

We apply our methodology to the specific problem of fault localization across IP and optical network layers, a difficult problem faced by network operators today. Currently, when IP operations receives router-interface alarms, the systems and staff are often faced with time-intensive manual investigation of what layer the problem occurred in, where, and why. This task is hampered by the architecture of the underlying network: IP uses optics for transport and (in some cases) for self-healing services (e.g., SONET ring restoration) in an overlay fashion. The

task of managing each of the two network layers is naturally separated into independent software systems.

Joining dynamic fault data across IP and optical systems is highly challenging—the network elements, supporting standards and information models are totally different. Though there are fields, such as circuit IDs, which can be used to join databases across these systems, automated mechanisms to assure the accuracy of these joins are lacking. Unfortunately, the network elements and protocols provide little help. Path-trace capabilities (counterparts of IP traceroute) are not available in the optical layer, or, if available, do not work in a multi-vendor environment (e.g., where the DWDM systems are provided by multiple vendors). In optical systems such as SONET, there is no counterpart to IP utilization statistics, which might be used to correlate traffic at the IP layer with the optical layer. Both IP and optical network topologies are rapidly changing as equipment is upgraded, network reach is extended, and capacities are re-engineered to manage changing demands.

Our key contribution is the novel and successful application of risk modeling to localize faults across the IP and optical layers in operational networks. Roughly speaking, a physical object such as a fiber span or an optical amplifier represents a *shared risk* for a group of logical entities (such as IP links) at the IP layer. That is, if the optical device fails or degrades, all of the IP components that had relied upon that object fail or degrade. In the literature, these associations are referred to as *Shared Risk Link Groups* or SRLGs [4]. Using only event data gathered at IP layer, and topology data gathered at both IP and optical layers, we bridge the gap between the operational information network managers need and what is actually reported at IP layer. SCORE (Spatial Correlation Engine) relieves operators of the burden of cross-correlating dynamic fault information from two disparate network layers. Once the layer and the location of the fault has been determined, other systems and tools at the appropriate layer can be targeted towards identifying the precise characteristics (for example, rule-based or statistical methods [12, 21]).

2 Troubleshooting using shared risks

Monitoring alarms associated with IP network component failures are typically generated on an individual basis—for example, a router failure will appear as a failure of all of the links terminating at that router. Best current practice requires a manual correlation of the individual link failure notifications to determine that they are all a result of a common network element (e.g., router). In more complicated failure scenarios, however, it is substantially more challenging to group individual alarms into common groups, and often difficult to even identify in which layer the fault occurred (e.g., in the transport

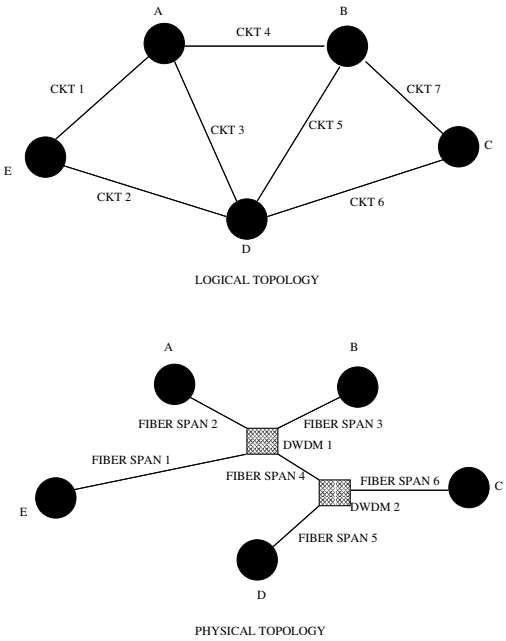


Figure 1: Example illustrating the concept of SRLGs.

network interconnecting routers, or in the routers themselves). By identifying the set of possible components that could have caused the observed symptoms, shared risk analysis can serve as the first step of diagnosing a network problem. For events being investigating by operations personnel in real time, reducing the time required for troubleshooting directly decreases down time.

2.1 Shared risk in IP networks

Our challenge is to construct a model of risks that represent the set of IP links that would likely be impacted by the failure of each component within the network. The tremendous complexity of the hardware and software upon which an IP network is built implies that constructing a model that accounts for every possible failure mode is impractical. Instead, we identify the key components of the risk model that represent the prevalent network failure modes and those that do not require deep knowledge of each vendor's equipment used within the network. We hasten to add that the better the SRLG modeling of the network, the more precise the fault diagnosis can be. However, as we show later, a solid SRLG model combined with a flexible spatial correlation algorithm can ensure that fault isolation can be robust to missing details in the risk model developed.

The basic network topology can be represented as a set of nodes interconnected via links. Inter-domain and intra-domain routing protocols such as OSPF and BGP operate with a basic abstraction of a point-to-point link between a routers. Of course, OSPF permits other abstractions such as multi-access and non-broadcast, but

a backbone network typically only consists of point-to-point links between routers. Figure 1 illustrates a very simplistic network consisting of five nodes connected via six optical links (circuits). Each inter-office IP link is carried on an optical circuit (typically using SONET). This optical circuit in turns consists of a series of one or more fibers, optical amplifiers, SONET rings, intelligent optical mesh networks and/or DWDM systems [18]. These systems consist of network elements that provide O-E-O (optical to electrical) conversion and, in the case of SONET rings or mesh optical networks, protection/restoration to recover from optical layer failures. Multiple optical fibers are then carried in a single conduit, commonly known as a *fiber span*. Typically, each optical component may carry multiple IP links—the failure of these components would result in the failure of all of these IP links. We illustrate this concept in the bottom half of Figure 1, where we show the optical layer topology over which the IP links are routed. In the Figure 1, these shared risks are denoted as FIBER SPAN 1 to 6, DWDM 1 and 2. CKT3 and CKT5 are both routed over FIBER SPAN 4 and thus would both fail with the failure of FIBER SPAN 4. Similarly DWDM 1 is shared between CKT 1, 3, 4 and 5, while CKT 6 and CKT 7 share DWDM 2.

In essence, each network element represents a shared risk among all the links that traverse through this element. Hence, this set of links represents what is known as the Shared Risk Link Group (SRLG), as defined in [14, 23]. This concept is well understood in the context of network planning where backup paths are chosen such that they do not have any SRLG in common with the primary path, and sufficient capacity is planned to survive SRLG failures. However, the application of risk group models to real-time and offline fault analysis has not been well explored.

2.2 Network SRLGs

We now present the shared risk group model that we construct to represent a typical IP network. We divide the model into hardware-related risks and software risks. Note that this model is not exhaustive, and can be expanded to incorporate, for example, additional software protocols.

2.2.1 Hardware-related SRLGs

Fiber: At the lowest level, a single optical fiber carries multiple wavelengths using DWDM. One or more IP links are carried on a given wavelength. All wavelengths that propagate through a fiber form an SRLG with the fiber being the risk element. A single fiber cut can simultaneously induce faults on all of the IP links that ride over that fiber.

Fiber span: In practice, a set of fibers are carried together through a cable. A set of cables are laid out in a conduit. A cut (from, e.g., a backhoe) can simultaneously fail all links carried through the conduit. These set of circuits that ride through the conduit form the fiber span SRLG.

SONET network elements: In practice SONET network elements such as optical amplifiers, add-drop multiplexors etc., are shared across multiple wavelengths (that represent the circuits). For example, an optical amplifier amplifies all the wavelengths simultaneously – hence a problem in the optical amplifier can potentially disrupt all associated wavelengths. We collectively group these elements together into the SONET network elements group.

Router modules: A router is usually composed of a set of modules, each of which can terminate one or more IP links. A module-related SRLG denotes all of the IP links terminating on the given module, as these would all be subject to failure should the module die.

Router: A router typically terminates a significant number of IP links, all of which would likely be impacted by a router failure (either software or hardware). Hence, all of the IP links terminating on a given router collectively belong to a given router SRLG.

Ports: An individual link can also fail due to the failure of a single port on the router (impacting only the one link), or through other failure modes that impact only the single link. Thus, we also include Port SRLGs in our model. Port SRLGs however are singleton sets consisting of only one circuit. However, we add them in our risk model in order to be able to explain single link failure.

2.2.2 Software-related SRLGs

Autonomous system: An autonomous system (AS) is a logical grouping of routers within the Internet or a single enterprise or provider network (typically managed by a common team and systems). These routers are typically all running a common instance of an intra-domain routing protocol and, although extremely rare, a single intra-domain routing protocol software implementation can cause an entire AS to fail.

OSPF areas: Although an OSPF area is a logical grouping of a set of links for intra-domain routing purposes, there can be instances where a faulty routing protocol implementation can cause disruptions across the entire area. Hence, the IP links in a particular area form an OSPF Area SRLG.

Not all SRLGs have corresponding failure diagnosis tools associated with them. For example, a fiber span is a physical piece of conduit that generally cannot indicate to the network operator that it has been cut. Similarly, there is no monitoring at the OSPF area level that can indicate if the whole area was affected. On the other hand, some SONET optical devices can indicate failures in real time. However, these failure indications are usu-

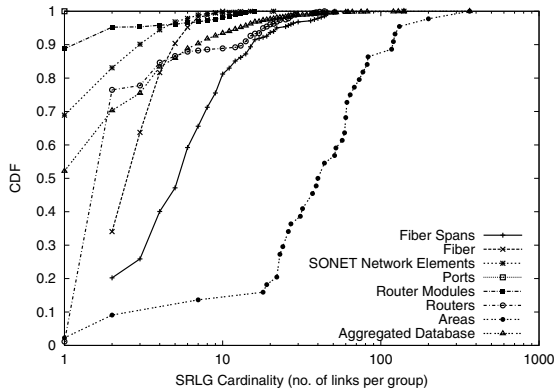


Figure 2: CDF of shared risk among real SRLGs

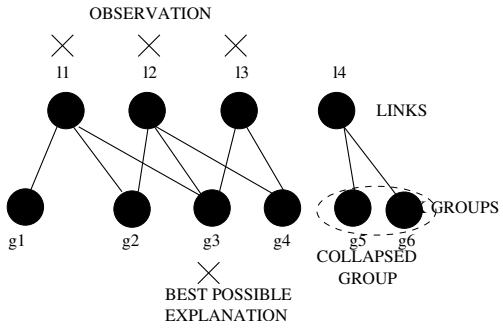


Figure 3: A bipartite graph formulation of the Shared Risk Group problem.

ally at wavelength granularity (i.e circuit or link level failures) and hence are not representative of that equipment failure. Diagnosis is therefore based on inference from correlated failures that can be attributed to a particular SRLG. In the absence of fault notifications directly from the equipment, this becomes the only approach to identify the failed component in the network.

2.3 Shared risk in real networks

Spatial correlation is inherently enabled by richness in sharing of risks between links. In particular, spatial correlation will typically be most effective in networks where SRLGs consist of multiple IP links, and each IP link consists of multiple SRLGs. Figure 2 depicts the cumulative distribution of the SRLG cardinality (the number of IP links in each SRLG) in a segment of a large tier-1 IP network backbone (in particular, customer-facing interfaces are not included here). The figure gives an idea of the SRLG cardinality (number of IP links per SRLG) in real-life. We can observe from this figure that, as expected, OSPF areas typically consist of a large number of links (and, hence, are included in their SRLG), whereas port SRLGs (by definition) comprise only a single circuit. In between, we can see that fiber spans typically have a significant number of IP links sharing them, while SONET network elements typically have fewer. The important observation here is that there is a significant degree of sharing of network components that can be utilized in spatial correlation in real IP networks. Studies of the number of SRLGs along each IP link show similar results. Thus, shared risk group analysis holds great promise for large-scale IP networks.

3 Shared Risk Group analysis

We begin by defining the notation we shall use throughout the remainder of the paper. Define an *observation* as a set of link failures that are potentially correlated, either temporally or otherwise. In other words, if a given set of links fail simultaneously or share a similar pattern or signature of a failure, these events form an observation. A *hypothesis* is a candidate set of circuit failures that could *explain* the observation. That is, a hypothesis is a set of risk groups that contain the set of links seen to fail in a given observation.

The goal, then, of shared risk group analysis is to obtain a hypothesis that best explains a given observation. The principle of Occam's razor suggests that the simplest explanation is the most likely; hence, we consider the best hypothesis to be the one with the fewest number of risk groups. We note, however, that there could be other formulations of the problem where a best hypothesis is optimizing some other metric.

3.1 Problem formulation

We can define the problem formally as follows. Given a set of links, $C = \{c_1, c_2, \ldots, c_n\}$, and risk groups $G = \{G_1, G_2, \ldots, G_m\}$. Each risk group $G_i \in G$ contains a set of links $G_i = \{c_{i1}, c_{i2}, \ldots, c_{ik}\} \subseteq C$ that are likely to fail simultaneously. (We use the terms "links" and "circuits" here to aid intuition, though it will be apparent that the formulation and the algorithm to be described simply deals with sets, and can be applied to arbitrary problem domains). Note that each circuit here can potentially belong to many different groups. Given an input observation consisting of events on a subset of circuits, $O = c_{e1}, c_{e2}, \ldots, c_{em}$, the problem is to identify the most probable hypothesis, $H = \{G_{h1}, G_{h2}, \ldots, G_{hk}\} \subseteq G$ such that H explains O, i.e., every member of O belongs to at least one member of H and all the members of a given group G_{hi} belong to O. The latter constraint stems from the fact that if a component fails, all the associated member links fail and hence should be a part of the observation. H is a set cover for O; finding a minimum set cover is known to be NP complete.

The problem can be modeled visually using a bipartite graph as shown in Figure 3. Each circuit, c_i, and group, G_j, is represented by a node in the graph. The

bottom partition consists of nodes corresponding to the risk groups; the top nodes correspond to circuits. An edge exists between a circuit node and a group node if that circuit is a member of the risk group. Given this bipartite graph and a subset of vertices in the top partition (corresponding to an observation), the problem is to identify the smallest possible set of group nodes that cover the events.

Before proceeding, we observe that if multiple risk groups have the same membership—that is, the same set of circuits may fail for two or more different reasons—it is impossible to distinguish between the causes. We call any such risk groups *aliases*, and collapse all identical groups into one in our set of risk groups. For example, in Figure 3, group $g5$ and $g6$ have the same membership: $l4$. Hence, $g5$ and $g6$ are collapsed into a single group as a pre-processing step.

3.2 Greedy approximation

There are potentially many different ways to solve the problem as formulated above; we use a greedy approximation to model imperfect fault notifications and other inconsistencies due to operational realities (as discussed in Section 3.3). Our greedy approximation also reduces the computation cost involved in identifying the most likely hypothesis among all hypotheses (which can potentially be large).

Before presenting the algorithm, however, we must first introduce two metrics we will use to quantify the utility of a risk group. Let $|G_i|$ be the total number of links that belong to the group G_i (known as the cardinality of G_i). Similarly, $|G_i \cap O|$ is the number of elements of G_i that also belong to O. We define the *hit ratio* of the group G_i as $|G_i \cap O|/|G_i|$. In other words, the hit ratio of a group is the fraction of circuits in the group that are part of the observation. The *coverage ratio* of a group G_i is defined as $|G_I \cap O|/|O|$. Basically, the coverage ratio is the portion of the observation explained by a given risk group.

Intuitively, our greedy algorithm attempts to iteratively select the risk group that explains the greatest number of faults in the observation with the least error: in other words, the highest coverage and hit ratios. More concretely, in every iteration, the algorithm computes the hit ratio and coverage ratio for all the groups that contain at least one element of the observation (i.e., the neighborhood of the observation in the bipartite graph). It selects the risk group with maximum coverage (subject to some restrictions on the hit ratio which we shall describe later) and prunes both the group and its member circuits from the graph. In the next iteration, the algorithm recomputes the hit and coverage ratio for the remaining set of groups and circuits. This process repeats, adding the group with the maximum coverage in each iteration to the hypothesis, until finally terminating when there are no circuits remaining in the observation. The pseudocode is presented in Algorithm 1.

Algorithm 1 greedyHypothesis(input_links,threshold)

```
1: explained = {}; // EmptySet
2: unexplained = input_links;
3: // All groups that contain at least one link
4: groups = getAllGroups(unexplained);
5: while (unexplained ≠ {}) do
6:    // Compute hit and coverage for all groups
7:    hitCoverage(groups, explained, unexplained);
8:    // Find a candidate group for pruning
9:    grp = findCandidateGroup(groups, threshold);
10:   pruneGrp(grp, explained, unexplained);
11:   addGroup(hypothesis, grp);
12: end while
13: return hypothesis;
```

Algorithm 2 findCandidateGroup(groups,threshold)

```
1: for all group such that group.hitratio ≥ threshold do
2:    maxGroup = updateMaxCoverage(group)
3: end for
4: return maxGroup
```

The algorithm maintains two separate lists: explained and unexplained. When a risk group is selected for inclusion in the hypothesis, all circuits in the observation that are explained by this risk group are removed from the unexplained list and placed in the explained list. The hit ratio is computed based on the union of the explained and unexplained list, but coverage ratio is computed based only on the unexplained list. The reason for this is straightforward: multiple failures of the same circuit will result in only one failure observation. Hence, the hit ratio of a risk group should not be reduced simply because some other risk group also accounts for the failure observation.

3.3 Modeling imperfections

In our discussion so far, we have skirted the issue of selecting risk groups with hit ratios less than one. What does it mean to have a hypothesis that explains more circuit failures than actually occurred? In a straightforward model, such a result is nonsensical: if the shared component generating the risk group failed, all constituent circuits should have been affected. Operational reality, however, is seemingly contradictory for a number of reasons, including incomplete or erroneous monitoring data, and inaccurate modeling of the shared risk groups.

The failure notices (e.g., SNMP traps) are often transmitted using unreliable protocols such as UDP which can

result in partial failure observations. Hence, the accuracy of the diagnosis can be impacted if the data is erroneous or incomplete. For example, if due to the failure of a particular optical component failure, six links went down out of which only five, say, messages made it to the monitoring system. The hit ratio for the risk group representing the shared component is then 5/6. Without expressly allowing for the selection of this risk group, the algorithm would output a hypothesis, that, while plausible, is likely far from reality.

Furthermore, while theoretically it should be possible to precisely model all risk groups, it is impossible in practice to exactly capture all possible failure modes. This difficulty leads to two interesting cases of inaccurate modeling. One is failure to model high-level risk groups (e.g., all links terminating in a particular point of presence may share a power grid) while the other is failure to model low-level risk groups (for example, some internal risk group within a router). Our algorithm needs to be robust against imprecise failure groups and, if possible, learn from real observations. We discuss one real instance of learning of new risk groups from actual failure observations in Section 6.

We allow for these operational realities by selecting the risk group with greatest coverage out of those with hit ratios above a certain error threshold. So, even if a particular circuit is omitted (either due to incorrect modeling or missing data), the error threshold allows consideration of groups that have most links but not quite all and cover a large number of failures.

Note that there could be two different cases once we include groups with hit ratios above a certain error threshold. It is possible that there are genuinely only few failures but due to information loss the algorithm with no error threshold outputs a hypothesis with larger number of failures. Relaxing the error threshold would account for this loss thereby outputting a better (smaller) hypothesis. On the other hand, there could be genuinely larger number of failures in which case relaxing the error threshold can output a wrong hypothesis.

It turns out to be extremely difficult to select a single error threshold for all observations, as it depends greatly on the size of individual risk groups involved in the observation. In practice, we run the algorithm multiple times and generate hypotheses for decreasing error thresholds until a plausible hypothesis is generated. More generally, we can assign a cost function to evaluate the confidence of a particular hypothesis based on the number of component failures in the hypothesis and the threshold used and choose the one with lowest cost.

4 System overview

We created SCORE with generality in mind. Accordingly, key systems and algorithmic components are factored out so that they may be reused in multiple problem domains or in variations for a single problem domain. A stand-alone *spatial correlation module* is driven by an extensible set of problem domain dependent diagnosis processes. Intelligence from the problem domain is built into the SRLG database, and is reflected in the SCORE queries. Figure 4 depicts the SCORE system architecture as it is implemented today. The following subsections describe the various modules in more detail.

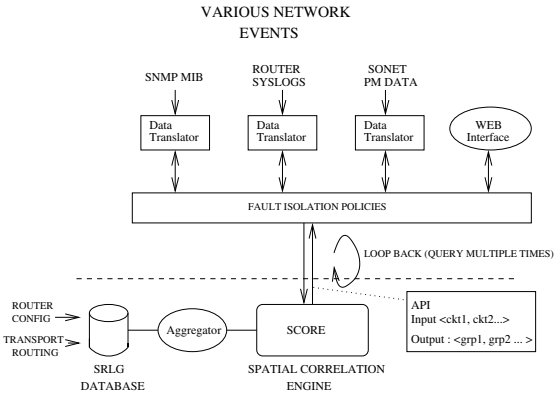


Figure 4: System architecture framework of SCORE.

4.1 SRLG database

The SRLG database manages relationships between SRLG groups and corresponding links. For example, in our application, the database atoms used to form SRLGs at SONET layer describe SONET level equipment IDs that particular IP links traverses, extracted from databases populated from operational optical element management systems. Other risk groups such as area, router, modules, etc. are similarly formed from the native databases extracted from the various network elements (e.g. router configurations). We note that the underlying databases track the network and therefore exhibit churn. The SCORE software is currently snapshot driven, and copes with churn by reloading multiple times during the course of the day. As mentioned in Section 3 on alias aggregation, we collapse risk groups with identical member links, prior to performing spatial correlation.

4.2 Spatial correlation engine

The Spatial Correlation Engine (SCORE) forms the core of the system. This engine periodically loads the spatial database hierarchy and responds to queries for fault localization. SCORE implements the greedy algorithm discussed in Section 3. That is, SCORE obtains the minimum set hypothesis using the SRLG database and a given set of inputs. Optionally, an error threshold can be specified, as described in Section 3.

4.3 Data sources

The set of observations upon which spatial correlation is applied are obtained from the network fault notifications and performance reports (including IP performance-related alarms). These in turn come from a wide range of data sources. We discuss below some of the more popular fault and performance-related data sources that have been used within the SCORE architecture to date. Though we describe certain optical layer event data sources (such as SONET PM data) and have experimented with such sources with SCORE, only IP event sources were used to obtain the results described in this paper.

4.3.1 IP layer Fault notifications

IP link failures and other faults will be observed by the routers, and reported to centralized network operations systems via SNMP traps sent from the router. These SNMP traps provide the key event notifications that allow network operators to learn of faults as they occur.

Router operating systems, much like Unix operating systems, log important events as they are observed. These are known as *router syslogs* and provide a wealth of useful information regarding network events. These can be used as additional information to complement the SNMP traps and the alarms that they generate. Table 1 shows sample Syslog messages for a failure observed on a Cisco router, and another failure observed on an Avici router. The failures are reported at different layers—illustrated here for the SONET layer, PPP layer and IP layer (OSPF). Note that there is no standardized format for these messages as they are usually output for debugging purposes.

4.3.2 Performance reports

SNMP performance data is generated by the routers on either a per-interface or per-router basis, as applicable. It typically contains 5 minute aggregate measurements of statistics such as traffic volumes, router CPU average utilization, memory utilization of the router, number of packet errors, packet discards and so on.

Performance metrics are also available on a per circuit basis from SONET network elements along an optical path (as are alarms, although these are not discussed here). Numerous parameters will be reported in, for example, 15 minute aggregates. These include parameters such as coding violations, errored seconds and severely errored seconds (indicative of bit error rates and outages), and protection switching counts on SONET rings.

4.4 Data translation/normalization

Each of these monitoring data are usually collected from different network elements (such as routers, SONET DWDM equipment etc.) and streamed to a centralized database. These different data are usually stored in different formats with different candidate keys. For example, the candidate key for SNMP database is an interface number as it collects interface-level statistics. OSPF messages are based on link IP addresses. SONET performance monitoring data is based on a circuit identifier. All these data sources are mapped into link circuit identifiers using a set of mapping databases.[1]

4.5 Fault localization policies

Fault localization is performed on various monitoring data sources (such as those mentioned in the previous section) using flexible data-dependent policies. In Figure 4, fault isolation policies form the bridge between the various monitoring data sources (translators) and the main SCORE engine. These policies dictate how a particular type of fault can be localized. The main functions include:

- **Event Clustering**. Clustering events that represent either temporally correlated events or events with similar failure signature (hence could be spatially correlated)
- **Localization Heuristics**. Heuristics that dictate how to identify the hypothesis that can best explain these event clusters.

Event Clustering: Data sources that are based on discrete asynchronous events, (e.g. OSPF messages, Syslog messages) need to be clustered to identify an observation. This clustering captures all the events that took place in a fixed time interval as potentially correlated. Note that a failure can have events that are slightly off in time either due to time synchronization issues across various elements, or propagation delays in an event to be recorded. Hence, event clustering has to account for these in recording observations.

There are many different ways to cluster events. A naive approach to clustering is based on fixed time bins. For example, we can make observations (set of links potentially correlated) by clustering together all events in a fixed 5 minute bin. The problem with this approach however is the fact that events related to a particular failure can potentially straddle the time bin boundary. In this case, this quantization will create two different observations for correlated events thus affecting the accuracy of the diagnosis.

In our system, we use a clustering algorithm based on gaps between failure events. We use the largest chain of events that are spaced apart within a set threshold (called quiet period) as potentially correlated events. The intuition here is that two events that occur within a time period less than a given threshold (say 30 seconds) are po-

[1] We map all the databases into link circuit identifiers since the network database itself is organized based on link circuit identifiers. However, any unified format would work equally well.

Syslog Message on Cisco/Avici Routers	Layer	Router
Aug 16 04:01:29.302 EDT: %LINEPROTO-5-UPDOWN: Line protocol on Interface POS0/0, changed state to down	SONET layer	Cisco
Aug 16 04:01:29.305 EDT: %LINK-3-UPDOWN: Interface POS0/0, changed state to down	PPP layer	Cisco
Aug 16 04:01:29.308 EDT: %OSPF-5-ADJCHG: Process 11, Nbr 1.1.1.1 On POS0/0 from FULL to DOWN, Neighbor Down: Interface down or detached	OSPF/IP layer	Cisco
module0036:SUN SEP 12 17:23:29 2004 [030042FF] MINOR:snmp-traps :Sonet link POS 1/0/0 has new adminStatus up and operStatus up.	Sonet Layer	Avici
server0001:SUN SEP 12 17:25:01 2004 [030042FF] MINOR:snmp-traps :PPP link POS 1/0/0 has new adminStatus up and operStatus up.	PPP layer	Avici
server0002:THU AUG 12 07:21:58 2004 [030042FF] MINOR:snmp-traps:OSPF with routerId 1.1.1.1 had non-virtual neighbor state change with neighbor 1.1.1.2 (address less 0) (router id 1.1.1.4) to state Down.	OSPF/IP layer	Avici

Table 1: Syslog messages output by Cisco and Avici routers when a link goes down at different layers of the stack. When the link comes back up, the router writes similar messages indicating that each of the layer is back up.

tentially correlated and can be attributed to the same failure. Note however that this particular parameter needs to be tuned for the particular problem domain. These clustered events are then fed to the SCORE system to obtain a hypothesis that represents the failed components in the network.

Although currently we use temporally correlated events as a good indication of events that potentially can have the same root cause, it is possible to apply different methods to cluster events. One such alternative method effective for software bugs where a particular type of router with a particular version of software might have fault signatures for different links although not necessarily temporally correlated. An offline analysis tool that can observe these signatures can then query and find out the risk group associated with these links.

Localization Heuristics: Fault localization often requires heuristics that are either derived intuitively or through domain knowledge to make multiple queries to the system with different parameters in order to obtain higher confidence hypothesis. SCORE architecture itself allows flexible overlay of such troubleshooting policies depending on the problem domain. We implemented one such localization heuristic for handling IP link down events (including possible database errors).

A simple heuristic we implemented is to query Spatial Correlation engine with multiple error thresholds (reducing from 1.0 to 0.5) and obtain many different hypotheses. We compare these hypotheses obtained using different relaxations (error thresholds) to account for data inconsistencies or database issues. The most likely hypothesis is based on a cost function that depends on the amount of error threshold, number of failures in the hypothesis and finally the individual types of groups in the hypothesis. This policy is more applicable to link failures identified at the IP layer. Currently we use the ratio between number of groups and the threshold; we would like to identify cases where a small relaxation in the threshold (say error threshold of 0.9) can reduce the number of groups significantly.

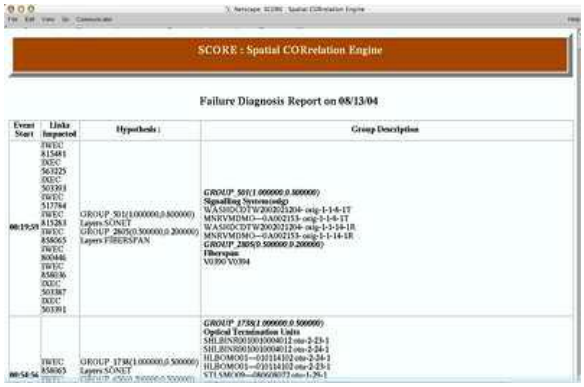


Figure 5: SCORE screen shot.

Another heuristic is to query the Spatial Correlation engine using clustered events that have similar signature (e.g. links that had same number of bit errors in a given time frame). This policy is guided by the intuition that correlated events in terms of the actual signature potentially have the same root cause. This heuristic is more suitable to diagnose root-causes of soft errors in SONET performance monitoring data.

4.6 Implementation issues

The main core engine loads (and periodically refreshes) an SRLG database that has associations from groups to a set of links. It constructs two hashtables, one for the set of circuits and one for the set of groups. Each group consists of the circuit identifiers that can be used to query the circuits hashtable. This particular implementation allows for fast associations and traversals to implement the spatial correlation algorithm outlined in Section 3. The total implementation of this main SCORE engine is slightly more than 1000 lines of C code. This engine also has a listening server at a particular port on which various diagnosis agents can connect via popular socket interface

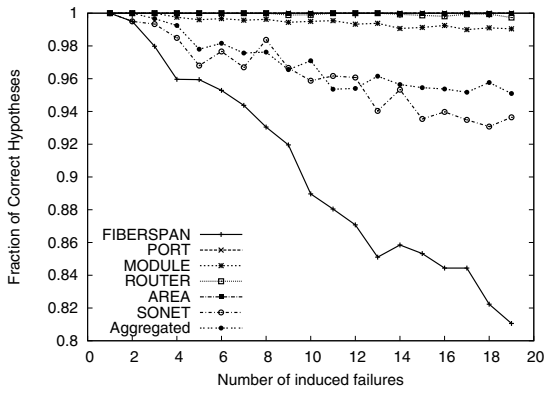


Figure 6: Percentage correct hypotheses as a function of error probability and for various algorithm error thresholds (three simultaneous failures).

and perform queries on the cluster of link-failures. The SCORE engine then responds with the hypotheses that can best explain the cluster.

SCORE system is not extremely difficult to implement. Obtaining groups from different databases that contain fiber level, fiber span level, router level and other independent databases is one of the functions of the SRLG database module. This module is implemented in perl and it consists of about 1000 lines of code. The other function of the SRLG database interface is the group alias resolution. This group alias resolution algorithm is not a performance bottleneck as it is refreshed fairly infrequently (usually twice a day) resolution algorithm in perl. This collapsing of risk groups itself is about 200 lines of perl code.

Clustering events and writing a per-data source event collection module is written in perl. Finally, the troubleshooting policies themselves need the flexibility and hence have been implemented in perl too. The total lines of code for the two policies we implemented is slightly less than a thousand.

The SCORE web interfaces consists of a table consisting of the following columns. Figure 5 shows a live screenshot of the SCORE web interface. The interface also allows to view archived logs including raw events and their associated diagnosis results. The first column outputs the actual event start time and the end time using one of the clustering algorithms. The second column represents the set of links that were impacted during the event. The third and fourth column give descriptions of the groups of components that form the diagnosis report for that observation. The diagnosis report also consists of the hit-ratio, coverage ratio and finally error threshold used for the groups involved in the diagnosis.

5 Simulated faults

We evaluated the performance of the SCORE spatial correlation algorithm using both artificially generated faults (this section) and real faults (next section). The main goal of the initial experiments is to evaluate the accuracy of the greedy approach within a controlled environment by using emulated faults. We use an SRLG database constructed from the network topology and configuration data of a tier-1 service provider's backbone. In our simulation, we inject different number of simultaneous faults into the system and evaluate the accuracy of the algorithm in obtaining the correct hypothesis. We first study the efficacy of the greedy algorithm under ideal operating conditions (no losses in data, no database inconsistencies) followed by the presence of noisy data by simulating errors in the SRLG database and observations.

5.1 Perfect fault notification

To evaluate the accuracy of the SCORE algorithm, we simulated scenarios consisting of multiple simultaneous failures and evaluated the accuracy in terms of the number of correct hypotheses (faults correctly localized by the algorithm) and the number of incorrect hypotheses (those which we did not successfully localize to the correct components). We randomly generated a given number of simultaneous failures selected from the set of all network risk groups: the set of all SONET components, fiber spans, OSPF areas, routers, and router ports and modules in our SRLG database. Once the faults were selected for a given scenario, we identified the union of all the links that belong to these failures. These link-level failures were then input to the SCORE system and hypotheses were generated. The resulting hypotheses were then compared with the actual injected failures to determine those which were correctly identified, and those which were not.

Figure 6 depicts the fraction of correctly identified hypotheses as a function of the number of injected faults, where each data point represents an average across 100 independent simulations. The figure illustrates that the accuracy of the algorithm on these data sets is greater than 99% for Ports, Modules and Routers, irrespective of the number of simultaneous failures generated. In general, the accuracy of the algorithm decreases as the number of simultaneous failures increases, although the accuracy remains greater than 95% for less than five simultaneous failures. In reality, it is unlikely that more than one failure will occur (and be reported) at a single point in time. Thus, for failures such as fiber cuts, router failures, and module outages (corresponding to a single simultaneous failure), our results indicate that the accuracy of the system is near 100%. However, it is entirely possible in a large network that multiple independent components will simultaneously be experiencing minor per-

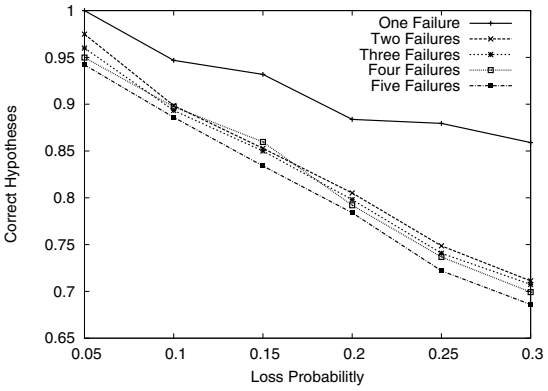


Figure 7: Percentage correct hypotheses as a function of error probability and for varying number of simultaneous failures (error threshold = 0.6).

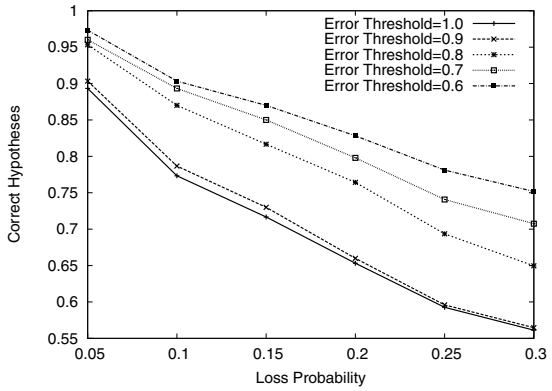


Figure 8: False hypotheses vs error probability (three failures).

formance degradations, such as error rates, which are reported and investigated on a longer time scale. Thus, the results representing higher number of simultaneous failures are likely indicative of performance troubleshooting. However, we can still conclude that for realistic network SRLGs, the greedy algorithm presented here is highly accurate when we have perfect knowledge of our SRLGs and failure observations.

5.2 Imperfect fault notification

The SRLG model provides a solid, but not perfect representation of the possible failure modes within a complex operational network. Thus, we expect to find scenarios where the set of observations cannot be perfectly described by any SRLG. Similarly, data loss associated with event notifications and database errors are inherent operational realities in managing large-scale IP backbones. In Section 3, we discussed how to adapt the basic greedy algorithm to account for these operational realities. In this section, we evaluate the accuracy of the SCORE algorithm when we have loss in our observations, which may result for example from imperfect event notifications (where failures are not reported for whatever reason). We consider three parameters: the error threshold used in the SCORE algorithm, the number of simultaneous failures, and the error probability (which represents the percentage of IP link failure notifications lost for a given failure scenario).

Figures 7 and 8 demonstrate the accuracy of the algorithm under a range of error probabilities and algorithm error thresholds and for different numbers of simultaneous failures. Specifically, the figures plot the percentage of correct hypotheses as a function of the error probability. In Figure 7, the algorithm error threshold is varied from 0.6 to 1.0, whilst the number of simultaneous failures is set to 3. In Figure 8 the algorithm error threshold is fixed at 0.6 and the number of simultaneous failures is varied from 1 to 5. As expected, increasing the error probability reduces the accuracy of the algorithm. Under three simultaneous failure events and an error probability of 0.1, we can observe from Figure 7 that an algorithm error threshold of between 0.7 and 0.8 restores the accuracy of the SCORE algorithm to around 90%. However, if we mandate perfect matching of failure observations to SRLGs (i.e., error threshold = 1.0), then our accuracy in isolating our fault drops to around 78%. This shows the necessity and effectiveness of the of the error thresholds introduced into the algorithm for fault localization in the face of noisy event observation data.

5.3 Performance results

The algorithm's execution time was also evaluated under a range of conditions. In general, the execution time recorded increased as the number of IP links (observations) impacted by the failures increased. This is because all of the SRLGs associated with each of the failed links must be included as part of the candidate set of SRLGs for localization, and thus must be evaluated. Thus, the execution time increased within increasing numbers of failures, but on average was below 150 ms for up to ten failures. Similarly, the execution time for scenarios involving router failures was typically higher than for other failure scenarios, as the routers typically involved larger numbers of links. Execution times of up to 400 ms were recorded for events involving large routers. However, even in these worst case scenarios, the algorithm is more than fast enough for real-time operational environments.

6 Experience in a tier-1 backbone

The SCORE prototype implementation was recently deployed in a tier-1 backbone network, and used in an offline fashion to isolate IP link failures reported in the network. The implemented system operated on a range of fault and performance data, including IP fault notifications and optical layer performance measures. However,

Type of problem	Component Name	#SRLGS (Thld.=1.0)	Final Thld	#SRLGS (Thld.=Final)	#Correctly localized	#Incorrectly localized	Comment
Router	Router A	27	0.8	1	1	0	No event reported by some links
Router	Router B	20	0.9	3	3	0	No event reported by some links
Router	Router C	12	0.7	1	1	0	No event reported by some links
Router	Router D	1	1	1	1	0	-
Router	Router E	18	0.8	1	1	0	No event reported by some links
Router	Router F	1	1	1	1	0	-
Router	Router G	4	1	4	4	0	-
Module	Module A	1	1	1	1	0	-
Module	Module B	1	1	1	1	0	-
Module	Module C	1	1	1	1	0	-
Optical	Sonet A	8	0.9	2	1	1	No observation reported by one link and database problem
Failed Transceiver	Sonet B	1	1	1	1	0	-
Short term Flap	Sonet C	2	0.7	1	1	0	No observation reported by one link
Optical Amplifier	Sonet D	2	0.6	1	1	0	No observation reported by one link
Fiber Cut	Fiber A	3	0.5	1	1	2	Database problem
Fiber Span	Fiber Span A	1	1	1	1	0	-
Protocol Bug	OSPF Area A	20	0.7	4	4	0	Incorrect SRLG modeling
Protocol Bug	OSPF Area A	4	1	4	4	0	OSPF Area A MPLS enabled interfaces

Table 2: Summary of real, tier-1 backbone failures successfully diagnosed by SCORE.

we limit our discussion here to our experience with link failure events reported in router syslogs.

Determining whether or not the SCORE prototype correctly localized a given fault requires identification of the root cause of the fault via other means. In many cases, identifying this root cause involved sifting through large amounts of data and reports—a tedious process at best. We were able to manually confirm the root cause of 18 faults; we present a comparison with the output reported by the SCORE prototype. We note, however, that our methodology has an inherent bias: we cannot exclude the possibility that there may be a correlation (not necessarily positive) between our ability to diagnose the fault and SCORE's performance. While it would have been preferable to select a subset of the faults at random, we were not able to manually diagnose every fault, nor did we have the resources to consider all faults experienced during SCORE's deployment.

Table 2 denotes the results of our analysis of each of our 18 faults. For each failure scenario, we report:

- the type of failure that occurred
- a name uniquely identifying the failed component
- the number of SRLG groups localized when the algorithm was run with a threshold of 1.0
- the threshold used to generate a final conclusion
- the number of SRLGs localized when the algorithm was run with the final threshold
- the number of SRLGs correctly localized
- the number of SRLGs incorrectly localized
- description of the reason why we had to reduce the threshold, or why we were unable to identify a single SRLG as the root cause in certain situations

Overall, we were able to successfully localize all of the faults studied to the SRLGs in which the failed network elements were classified—except where we encountered errors in our SRLG database. However, when we used a threshold of 1.0 (i.e., mandated that an SRLG can be identified if and only if faults were observed on all IP links), then we were typically unsuccessful—particularly for router failures, and for the protocol bug reported. In the majority of the router failures, even though these events corresponded to routers being rebooted, the remote ends of the links terminating on these routers did not always report associated link-level events. This may be due to a number of possible scenarios: the events may never have been logged in the syslogs, data may have been lost from the syslogs, the links may have been operationally shut down and hence did not fail at this point in time, or the links were not impacted by the reboot. Independent of why the link notifications were not always observed, the router failures were all successfully localized when the threshold was marginally reduced. This highlights the importance of the threshold concept in the SCORE algorithm to localize faults in operational networks.

Of course, router failures are typically easy to identify

through spatial correlation, as all of the links impacted have a common end point (the failed router). However, optical layer impairments can impact seemingly logically independent links at the IP layer if these links are all routed through a common optical component, making them much more difficult to identify.

We study four different SONET network element failures. The first—an optical amplifier failure—induced faults reported on 13 IP links. Thus, with a threshold of 1.0 our algorithm identified 8 different SRLGs as being involved. However, as the threshold was reduced to 0.9, we were able to isolate the fault to only 2 different SRLGs. Further reductions in this threshold did not, however, further reduce the number of SRLGs to which the fault was localized. Further investigation uncovered an SRLG database problem—where our SONET network element database did not contain any information regarding one of the circuits reporting the fault. Thus, the SCORE algorithm was unable to localize the fault for this particular IP link to the SRLG containing the failed optical amplifier, and instead incorrectly concluded that a router port was also involved (the second SRLG). However, the SRLG containing the failed amplifier was also correctly identified for the other 12 IP links—the lower threshold was required because no fault notification was observed for one of the IP links routed through the optical amplifier.

This optical amplifier example highlights a particularly important capability of the SCORE system—the ability to highlight potential SRLG database errors. Links missing from databases, incorrect optical layer routing information regarding circuits and other potential errors in databases play havoc with capacity planning and network operations and so must be identified. In this scenario, the database error was highlighted by the fact that we were unable to identify a single SRLG for a single network failure, even after lowering threshold using in the SCORE algorithm.

The other three SONET failures were all correctly isolated to the SRLG containing the failed network element, in two cases we again had to lower the threshold used within the algorithm to account for links for which we had no failure notification (in one of these cases, the missing link was indeed a result of the interface having been operationally shut down before the failure).

We tested our SCORE prototype on a second, previously identified failure scenario impacted by a SRLG database error (fiber A in table 2). Again, the SCORE system was unable to identify a single SRLG as being the culprit even as the threshold was lowered—as no SRLG in the database contained all of the circuits reporting the fault. So again, a database error was highlighted by the inability of the system to correlate the failure to a single SRLG.

The final case that we evaluated was one in which a low level protocol implementation problem (commonly known as a software bug!) impacted a number of links within a common OSPF area. This scenario occurred over an extended period of time, during which three other independent failures were simultaneously observed in other areas.

When a threshold of 1.0 was used in the SCORE algorithm, the event in question was identified as being the result of 20 independent SRLG failures—a large number even for the extended period of time! As the threshold was reduced to a final value of 0.7, the event was isolated to four individual SRLGs—three SRLGs in other OSPF areas (corresponding to the independent failures) and the OSPF area in question. Thus, the SCORE algorithm was correctly able to identify that the event corresponded to a common OSPF area. However, further investigation uncovered that the reason why not all links in the OSPF area were impacted was that only those interfaces that were currently MPLS-enabled were affected. Thus, an additional SRLG was added to our SRLG database that incorporated the links in a given area that were MPLS enabled—application of this enhanced SRLG database successfully localized all of the SRLGs impacted by the four simultaneous failures with a threshold of 1.0. Thus, this illustrates how the threshold used in the SCORE algorithm can allow our results to be robust to incomplete modeling of all of the possible SRLGs—any level of modeling of risk groups can be inadequate as there could be more complicated failure scenarios that cannot be modeled by humans perfectly *a priori*. However, we also illustrated how we can continually learn new SRLGs through further analysis of new failure scenarios, thereby enhancing our SRLG modeling.

6.1 Localization Efficiency

While the 18 faults we have studied demonstrate the ability of SCORE to correctly localize faults, it does not give an indication about how much we could localize. In this section, we evaluate the efficiency of SCORE using a metric we call localization efficiency. The localization efficiency of a given observation is defined as the ratio of the number of components after localization to that before localization. In other words, it is the fraction of components that are likely to explain a particular fault (or observation) using our localization algorithm out of all the components that can cause a given fault.

Define $G_i = \{G_{i1}, G_{i2}, \cdots, G_{ik}\}$ as the set of groups that a circuit c_i belongs to. O is an observation consisting of circuits $c_1, c_2, \cdots, c_i$. The most probable hypothesis $H \subseteq \cup_{k=0}^{i} G_k$ is the hypothesis output by the algorithm. Then, localization efficiency is given by $|H|/|\cup_{k=0}^{i} G_k|$.

In Figure 9, the cumulative distribution function of the localization efficiency is shown. From the Figure 9, we

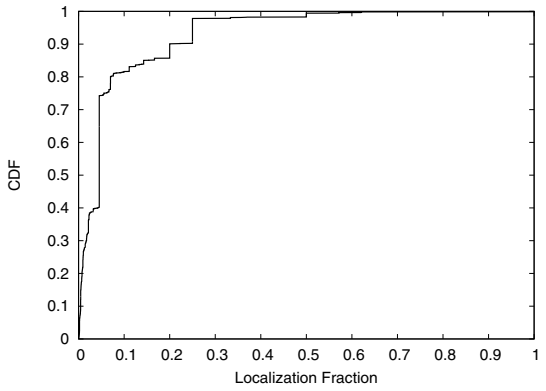


Figure 9: CDF of localization efficiency out of about 3000 real faults we have been able to localize.

can clearly observe that SCORE could localize faults to less than 5% for more than 40% of the failures and to less than 10% for more than 80% of the failures. This clearly demonstrates that SCORE can identify likely root causes very efficiently out of a large set of possible causes for a given failure.

7 Related work

Network engineers commonly employ the concept of Shared Risk Link Groups (SRLGs) to disjoint paths in optical networks, and serve as a key input into many traffic engineering mechanisms and protocols such as Generalized Multi-Protocol Label Switching (GMPLS). Due to their importance, there has been a great deal of recent work on automatically inferring SRLGs [20]. To the best of our knowledge, however, we are the first to use SRLGs in combination with IP-layer fault notifications to isolate failures in the optical hardware of a deployed network backbone without the need for monitoring at the physical layer.

Monitoring and management is a challenging problem for any large network. It is not surprising, then, that a number of research prototypes [17, 3, 15, 16, 6, 10] and commercial products have been developed to diagnose problems in IP and telephone networks. Commercial network fault management systems such as SMARTS [21], OpenView [12], IMPACT [13], EXCpert [16], and NetFACT [11] provide powerful, generic frameworks for handling fault indicators, particularly diverse SNMP-based [2] measurements, and rule-based correlation capabilities. These systems present a unified reporting interfaces to operators and other production network management systems. In general, however, they correlate alarms from a particular layer in order to isolate problems at that same layer (e.g., route flapping, circuit failure, etc.).

Roughan *et. al.* propose a correlation-based approach to detect forwarding anomalies including BGP-related anomalies [19]. Their approach was to detect events of potential interest by correlating multiple data sources while our approach was to diagnose these events to identify root causes.

The problem of fault isolation is obviously not limited to networking; similar problems exist in any complex system. Regardless of domain, fault detection systems have taken three basic approaches: rule or model-based reasoning [12, 1, 7], codebook approaches [21, 25], or machine learning (such as Bayesian or Belief Networks [24, 22, 5]). The difficulty with probabilistic or machine learning approaches is that they are not prescriptive: it's not clear what sets of scenarios they can handle besides the specific training data. Rule-based and codebook systems (otherwise known as "expert systems") are often even more specific, only being able to diagnose events that are explicitly programmed. Model-based approaches are more general, but require detailed information about the system under test. Dependency-based systems like ours, on the other hand, allow general inference without requiring undue specificity. Indeed, the specific use of dependency graphs for problem diagnosis has been explored before [9], but not in this particular domain.

Our problem as defined in Section 3.1 falls into the more general class of inference problems which include problems in other domains such as traffic matrix estimation, tomography, etc. Hence, techniques applied in these domains can be potentially used to solve this problem. For example, in [26], the authors reduce the problem of traffic matrix estimation to an ill-posed linear inverse problem and apply a regularization technique to estimate the traffic matrix. Similarly, our problem also can be solved using matrix inversion methods, using an incidence matrix to model the risks. While these methods can work well with perfect data, it is unclear how to adapt these techniques to deal with imperfect loss notifications and SRLG database errors. Besides, our greedy approximation works with an accuracy of over 95% for a large class of failures as shown in our evaluation (Section 5.1). Hence, the additional benefit in applying any other technique to solve the problem is only marginal.

8 Conclusions

Using our risk modeling methodology, we have developed a system that accurately localizes failures in an IP-over-optical tier-1 backbone. Given a set of IP-layer events occurring within a small time window, our heuristics pinpoint the shared risk (optical device) that best explains these events. Given the harsh operational reality of maintaining complex associations between objects in the two networking layers in separate databases, we find that it is necessary to go beyond identifying the single best explanation, and, instead, to generate a set of likely

explanations in order to be robust to transient database glitches.

We put forward a simple, threshold-based scheme that looks for best explanations admitting inconsistencies in the data underlying the explanations up to a given threshold. We find that not only does this increase the accuracy and robustness of fault localization, it also provided a new capability for identifying topology database problems, for which we had no alternative automated means of detecting. Getting shared risk information right is critical to IP network design. For example, a misidentification of a shared risk might produce a design believed to be resilient to single SRLG failure which in fact is not.

References

[1] S. Brugbosi, G. Bruno, et al. An Expert System for Real-Time Fault Diagnosis of the Italian Telecommunications Network. In *Proc. 3rd International Symposium on Integrated Network Management*, pages 617–628, 1993.

[2] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A Simple Network Management Protocol (SNMP). In *RFC 1157*, May 1990.

[3] C. S. Chao, D. L. Yang, and A. C. Liu. An automated fault diagnosis system using hierarchical reasoning and alarm correlation. *Journal of Network and Systems Management*, 9(2):183–202, 2001.

[4] S. Chaudhuri, G. Hjalmtysson, and J. Yates. Control of lightpaths in an optical network, Jan. 2000. IETF draft-chaudhuri-ip-olxc-control-00.txt.

[5] M. Chen, A. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer. A statistical learning approach to failure diagnosis. In *Proc. International Conference on Autonomic Computing (ICAC-04)*, May 2004.

[6] R. H. Deng, A. A. Lazar, and W.Wang. A probabilistic approach to fault diagnosis in linear lightwave networks. In *Integrated Network Management III*, pages 697–708, Apr. 1993.

[7] G. Forman, M. Jain, M. Mansouri-Samani, J. Martinka, and A. C. Snoeren. Automated whole-system diagnosis of distributed services using model-based reasoning. In *Proc. 9th IFIP/IEEE Workshop on Distributed Systems: Operations and Management*, Oct. 1998.

[8] GenSym. Integrity. `http://www.gensym.com`.

[9] B. Gruschke. Integrated event management: Event correlation using dependency graphs. In *Proc. 9th IFIP/IEEE Workshop on Distributed Systems: Operations and Management*, Oct. 1998.

[10] P. Hong and P. Sen. Incorporating non-deterministic reasoning in managing heterogeneous network. In *Integrated Network Management II*, pages 481–492, Apr. 1991.

[11] K. Houck, S. Calo, and A. Finkel. Towards a Practical Alarm Correlation System. In *Proc. 4th IEEE/IFIP Symposium on Integrated Network Management*, 1995.

[12] HP Technologies Inc. Open View. `http://www.openview.hp.com`.

[13] G. Jakobson and M. D. Weissman. Alarm correlation. *IEEE Network*, 7(6):52–59, Nov. 1993.

[14] I. P. Kaminow and T. L. Koch. *Optical Fiber Telecommunications IIIA*. Academic Press, 1997.

[15] G. Liu, A. K. Mok, and E. J. Yang. Composite events for network event correlation. In *Integrated Network Management VI*, pages 247–260, 1999.

[16] Y. A. Nygate. Event correlation using rule and object based techniques. In *Integrated Network Management, IV*, pages 278–289.

[17] P.Wu, R. Bhatnagar, L. Epshtein, M. Bhandaru, and Z. Shi. Alarm correlation engine (ACE). In *Proc. Network Operation and Management Symposium*, pages 733–742, 1998.

[18] R. Ramaswami and K. Sivarajan. *Optical Networks: A Practical Perspective*. Academic Press/Morgan Kaufmann, Feb. 1998.

[19] M. Roughan, T. Griffin, Z. M. Mao, A. Greenberg, and B. Freeman. Combining routing and traffic data for detection of ip forwarding anomalies. In *Proc. ACM SIGCOMM NeTs Workshop*, Aug. 2004.

[20] P. Sebos, J. Yates, D. Rubenstein, and A. Greenberg. Effectiveness of shared risk link group autodiscovery in optical networks. In *Proc. Optical Fiber Communications Conference.*, Mar. 2002.

[21] SMARTS. InCharge. `http://www.smarts.com`.

[22] M. Steinder and A. Sethi. End-to-end Service Failure Diagnosis Using Belief Networks. In *Proc. Network Operation and Management Symposium*, Florence, Italy, Apr. 2002.

[23] J. Strand, A. Chiu, and R. Tkach. Issues for routing in the optical layer. In *IEEE Communications*, Feb. 2001.

[24] H. Wietgrefe, K. Tochs, et al. Using Neural Networks for Alarm Correlation in Cellular Phone Networks. In *Proc. International Workshop on Applications of Neural Networks in Telecommunications*, 1997.

[25] S. A. Yemini, S. Kliger, E. Mozes, Y. Yemini, and D. Ohsie. High speed and robust event correlation. *IEEE Communications*, 34(5):82–90, 1996.

[26] Y. Zhang, M. Roughan, C. Lund, and D. Donoho. An Information-Theoretic Approach to Traffic Matrix Estimation. In *Proc. ACM SIGCOMM*, Aug. 2003.

Performance Modeling and System Management for Multi-component Online Services*

Christopher Stewart and Kai Shen
Department of Computer Science, University of Rochester
{stewart, kshen}@cs.rochester.edu

Abstract

Many dynamic-content online services are comprised of multiple interacting components and data partitions distributed across server clusters. Understanding the performance of these services is crucial for efficient system management. This paper presents a profile-driven performance model for cluster-based multi-component online services. Our offline constructed application profiles characterize component resource needs and inter-component communications. With a given component placement strategy, the application profile can be used to predict system throughput and average response time for the online service. Our model differentiates remote invocations from fast-path calls between co-located components and we measure the network delay caused by blocking inter-component communications. Validation with two J2EE-based online applications show that our model can predict application performance with small errors (less than 13% for throughput and less than 14% for the average response time). We also explore how this performance model can be used to assist system management functions for multi-component online services, with case examinations on optimized component placement, capacity planning, and cost-effectiveness analysis.

1 Introduction

Recent years have witnessed significant growth in online services, including Web search engines, digital libraries, and electronic commerce. These services are often deployed on clusters of commodity machines in order to achieve high availability, incremental scalability, and cost effectiveness [14, 16, 29, 32]. Their software architecture usually comprises multiple components, some reflecting intentionally modular design, others developed independently and subsequently assembled into a larger application, *e.g.*, to handle data from independent sources. A typical service might contain components responsible for data management, for business logic, and for presentation of results in HTML or XML.

*This work was supported in part by the National Science Foundation grants CCR-0306473, ITR/IIS-0312925, and CCF-0448413.

Previous studies have recognized the value of using performance models to guide resource provisioning for on-demand services [1, 12, 27]. Common factors affecting the system performance include the application characteristics, workload properties, system management policies, and available resources in the hosting platform.

However, the prior results are inadequate for predicting the performance of multi-component online services, which introduce several additional challenges. First, various application components often have different resource needs and components may interact with each other in complex ways. Second, unlike monolithic applications, the performance of multi-component services is also dependent upon the component placement and replication strategy on the hosting cluster. Our contribution is a comprehensive performance model that accurately predicts the throughput and response time of cluster-based multi-component online services.

Our basic approach (illustrated in Figure 1) is to build application profiles characterizing per-component resource consumption and inter-component communication patterns as functions of input workload properties. Specifically, our profiling focuses on application characteristics that may significantly affect the service throughput and response time. These include CPU, memory usage, remote invocation overhead, inter-component network bandwidth and blocking communication frequency. We perform profiling through operating system instrumentation to achieve transparency to the application and component middleware. With a given application profile and a component placement strategy, our model predicts system throughput by identifying and quantifying bottleneck resources. We predict the average service response time by modeling the queuing effect at each cluster node and estimating the network delay caused by blocking inter-component communications.

Based on the performance model, we explore ways that it can assist various system management functions for online services. First, we examine component placement and replication strategies that can achieve high performance with given hardware resources. Additionally, our model can be used to estimate resource needs to support projected future workload or to analyze the cost-

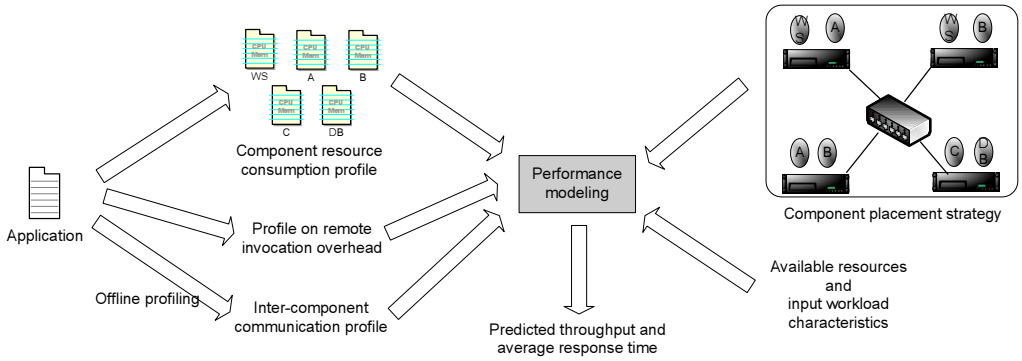


Figure 1: Profile-dri ven performance modeling for an online service containing a front-end Web server, a back-end database, and several middle-tier components (A, B, and C).

effectiveness of hypothetical hardware platforms. Although many previous studies have addressed various system management functions for online services [5, 6, 10, 31], they either deal with single application components or they treat a complex application as a non-partitionable unit. Such an approach restricts the management flexibility for multi-component online services and thus limits their performance potentials.

The rest of this paper is organized as follows. Section 2 describes our profiling techniques to characterize application resource needs and component interactions. Section 3 presents the throughput and response time models for multi-component online services. Section 4 validates our model using two J2EE-based applications on a heterogeneous Linux cluster. Section 5 provides case examinations on model-based component placement, capacity planning, and cost-effectiveness analysis. Section 6 describes related work. Section 7 concludes the paper and discusses future work.

2 Application Profiling

Due to the complexity of multi-component online services, an accurate performance model requires the knowledge of application characteristics that may affect the system performance under any possible component placement strategies. Our application profile captures three such characteristics: per-component resource needs (Section 2.1), the overhead of remote invocations (Section 2.2), and the inter-component communication delay (Section 2.3).

We perform profiling through operating system instrumentation to achieve transparency to the application and component middleware. Our only requirement on the middleware system is the flexibility to place components in ways we want. In particular, our profiling does not modify the application or the component middleware. Further, we treat each component as a blackbox (*i.e.*, we do not require any knowledge of the inner-working of application components). Although instrumentation at the application or the component middleware layer can more easily identify application characteristics, our OS-level approach has wider applicability. For instance, our approach remains effective for closed-source applications and middleware software.

2.1 Profiling Component Resource Consumption

A component profile specifies component resource needs as functions of input workload characteristics. For time-decaying resources such as CPU and disk I/O bandwidth, we use the average rates θ_{cpu} and θ_{disk} to capture the resource requirement specification. On the other hand, variation in available memory size often has a severe impact on application performance. In particular, a memory deficit during one time interval cannot be compensated by a memory surplus in the next time interval. A recent study by Doyle *et al.* models the service response time reduction resulting from increased memory cache size for Web servers serving static content [12]. However, such modeling is only feasible with intimate knowledge about application memory access pattern. To maintain the general applicability of our approach, we use the peak usage θ_{mem} for specifying the memory requirement in the component profile.

The input workload specifications in component profiles can be parameterized with an average request arrival rate $\lambda_{\text{workload}}$ and other workload characteristics δ_{workload} including the request arrival burstiness and the composition of different request types (or request mix). Putting these altogether, the resource requirement profile for a distributed application component specifies the following mapping $\xrightarrow{f}$:

$$(\lambda_{\text{workload}}, \delta_{\text{workload}}) \xrightarrow{f} (\theta_{\text{cpu}}, \theta_{\text{disk}}, \theta_{\text{mem}})$$

Since resource needs usually grow by a fixed amount with each additional input request per second, it is likely for θ_{cpu}, θ_{disk}, and θ_{mem} to follow linear relationships with $\lambda_{\text{workload}}$.

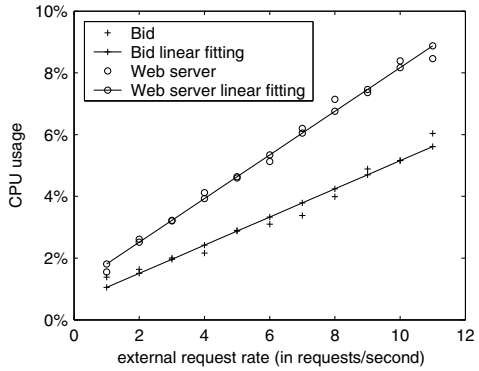


Figure 2: Linear fitting of CPU usage for two RUBiS components.

Component	CPU usage (in percentage)
Web server	$1.525 \cdot \lambda_{\text{workload}} + 0.777$
Database	$0.012 \cdot \lambda_{\text{workload}} + 3.875$
Bid	$0.302 \cdot \lambda_{\text{workload}} + 0.807$
BuyNow	$0.056 \cdot \lambda_{\text{workload}} + 0.441$
Category	$0.268 \cdot \lambda_{\text{workload}} + 1.150$
Comment	$0.079 \cdot \lambda_{\text{workload}} + 0.568$
Item	$0.346 \cdot \lambda_{\text{workload}} + 0.588$
Query	$0.172 \cdot \lambda_{\text{workload}} + 0.509$
Region	$0.096 \cdot \lambda_{\text{workload}} + 0.556$
User	$0.403 \cdot \lambda_{\text{workload}} + 0.726$
Transaction	$0.041 \cdot \lambda_{\text{workload}} + 0.528$

Table 1: Component resource profile for RUBiS (based on linear fitting of measured resource usage at 11 input request rates). $\lambda_{\text{workload}}$ is the average request arrival rate (in requests/second).

We use a modified Linux Trace Toolkit (LTT) [38] for our profiling. LTT instruments the Linux kernel with trace points, which record events and forward them to a user-level logging daemon through the `relayfs` file system. We augmented LTT by adding or modifying trace points at CPU context switch, network, and disk I/O events to report statistics that we are interested in. We believe our kernel instrumentation-based approach can also be applied for other operating systems. During our profile runs, each component runs on a dedicated server and we measure the component resource consumption at a number of input request rates and request mixes.

Profiling Results on Two Applications We present profiling results on two applications based on Enterprise Java Beans (EJB): the RUBiS online auction benchmark [9, 28] and the StockOnline stock trading application [34]. RUBiS implements the core functionality of an auction site: selling, browsing, and bidding. It follows the three-tier Web service model containing a front-end Web server, a back-end database, and nine movable business logic components (`Bid`, `BuyNow`, `Category`, `Comment`, `Item`, `Query`, `Region`, `User`, and `Transaction`). StockOnline contains a front-end Web server, a back-end database, and five EJB components (`Account`, `Item`, `Holding`, `StockTX`, and `Broker`).

Component	CPU usage (in percentage)
Web server	$0.904 \cdot \lambda_{\text{workload}} + 0.779$
Database	$0.008 \cdot \lambda_{\text{workload}} + 4.832$
Account	$0.219 \cdot \lambda_{\text{workload}} + 0.789$
Item	$0.346 \cdot \lambda_{\text{workload}} + 0.781$
Holding	$0.268 \cdot \lambda_{\text{workload}} + 0.674$
StockTX	$0.222 \cdot \lambda_{\text{workload}} + 0.490$
Broker	$1.829 \cdot \lambda_{\text{workload}} + 0.533$

Table 2: Component resource profile for StockOnline (based on linear fitting of measured resource usage at 11 input request rates). $\lambda_{\text{workload}}$ is the average request arrival rate (in requests/second).

Profiling measurements were conducted on a Linux cluster connected by a 1 Gbps Ethernet switch. Each profiling server is equipped with a 2.66 GHz Pentium 4 processor and 512 MB memory. For the two applications, the EJB components are hosted on a JBoss 3.2.3 application server with an embedded Tomcat 5.0 servlet container. The database server runs MySQL 4.0. The dataset for each application is sized according to database dumps published on the benchmark Web sites [28, 34].

After acquiring the component resource consumption at the measured input rates, we derive general functional mappings using linear fitting. Figure 2 shows such a derivation for the `Bid` component and the Web server in RUBiS. The results are for a request mix similar to the one in [9] (10% read-write requests and 90% read-only requests). The complete CPU profiling results for all 11 RUBiS components and 7 StockOnline components are listed in Table 1 and Table 2 respectively. We do not show the memory and disk I/O profiling results for brevity. The memory and disk I/O consumption for these two applications are relatively insignificant and they never become the bottleneck resource in any of our test settings.

2.2 Profiling Remote Invocation Overhead

Remote component invocations incur CPU overhead on tasks such as message passing, remote lookup, and data marshaling. When the interacting components are co-located on the same server, the component middleware often optimizes away these tasks and some even implement local component invocations using direct method calls. As a result, the invocation overhead between two components may vary depending on how they are placed on the hosting cluster. Therefore, it is important to identify the remote invocation costs such that we can correctly account for them when required by the component placement strategy.

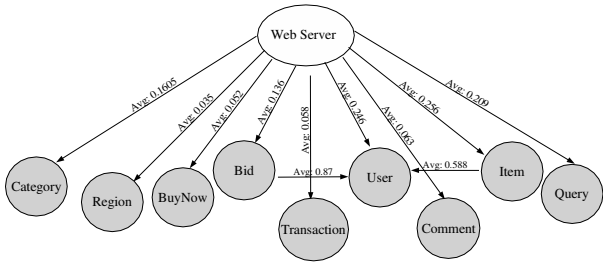


Figure 3: Profile on remote invocation overhead for RUBiS. The label on each edge indicates the per-request mean of inter-component remote invocation CPU overhead (in percentage).

It is challenging to measure the remote invocation overhead without assistance from the component middleware. Although kernel instrumentation tools such as LTT can provide accurate OS-level statistics, they do not directly supply middleware-level information. In particular, it is difficult to differentiate CPU usage of normal component execution from that of passing a message or serving a remote lookup query. We distinguish the remote invocation cost of component A invoking component B in a three step process. First, we isolate components A and B on separate machines. Second, we intercept communication rounds initiated by component A. We define a communication round as a sequence of messages between a pair of processes on two different machines in which the inter-message time does not exceed a threshold. Finally, we associate communication rounds with invocations. Thus, the remote invocation cost incurred between components A and B is the sum of resource usage during communication rounds between them. Since the components are isolated during the profiling, communication rounds are not likely to be affected by network noises.

Profiling Results Figure 3 shows the application profile on remote invocation overhead for RUBiS.

2.3 Profiling Inter-Component Communications

We profile component interaction patterns that may affect bandwidth usage and network service delay between distributed application components. We measure inter-component bandwidth consumption by intercepting all network messages between components during off-line profile runs. Note that the bandwidth usage also depends on the workload level, particularly the input user request rate. By measuring bandwidth usage at various input request rates and performing linear fitting, we can acquire per-request communication data volume for each inter-component link.

The processing of a user request may involve multiple blocking round-trip communications (corresponding to request-response rounds) along many inter-component links. We consider the request processing network delay as the sum of the network delays on all inter-component links between distributed components. The delay on each link includes the communication latency and the data transmission time. Inter-component network delay depends on the link round-trip latency and the number of blocking round-trip communications between components. We define a *round trip* as a synchronous write-read interaction between two components.

Due to the lack of knowledge on the application behavior, it is challenging to identify and count blocking round trip communications at the OS level. Our basic approach is to intercept system calls involving network reads and writes during profile runs. System call interception provides the OS-level information nearest to the application. We then count the number of consecutive write-read pairs in the message trace between two components. We also compare the timestamps of consecutive write-read pairs. Multiple write-read pairs occurring within a single network round-trip latency in the profiling environment are counted as only one. Such a situation could occur when a consecutive write-read pair does not correspond to each other in a blocking interaction. Figure 4 illustrates our approach for identifying blocking round trip communications. To avoid confusion among messages belonging to multiple concurrent requests, we process one request at a time during profile runs.

Profiling Results Our profiling results identify the per-request communication data volume and blocking round-trip interaction count for each inter-component link. Figure 5 shows inter-component communication profiles for RUBiS. The profiling result for each inter-component link indicates the per-request mean of profiled target.

2.4 Additional Profiling Issues

Non-Linear functional mappings. In our application studies, most of the workload parameter (*e.g.*, request arrival rate) to resource consumption mappings follow linear functional relationships. We acknowledge that such mappings may exhibit non-linear relationships in some cases, particularly when concurrent request executions affect each other's resource consumption (*e.g.*, extra CPU cost due to contention on a spin-lock). However, the framework of our performance model is equally applicable in these cases, as long as appropriate functional mappings can be extracted and included in the application profile. Non-linear fitting algorithms [4, 30] can be used for such a purpose. Note that a non-linear functional mapping may require more workload samples to produce an accurate fitting.

Profiling cost. To improve measurement accuracy, we place each profiled component on a dedicated machine.

```
#128 <1.732364sec> NET SEND: PID:13661 HOST_ADDR -> 128.151.67.29:41800 REMOTE_ADDR -> 128.151.67.228:8080 SIZE:177
#129 <1.734737sec> NET RECV: PID:13661 HOST_ADDR -> 128.151.67.29:41800 REMOTE_ADDR -> 128.151.67.228:8080 SIZE:1619
#130 <1.736060sec> NET RECV: PID:13661 HOST_ADDR -> 128.151.67.29:41800 REMOTE_ADDR -> 128.151.67.228:8080 SIZE:684
#131 <1.738076sec> NET RECV: PID:13661 HOST_ADDR -> 128.151.67.29:41800 REMOTE_ADDR -> 128.151.67.228:8080 SIZE:1448
#132 <1.738398sec> NET SEND: PID:13661 HOST_ADDR -> 128.151.67.29:41800 REMOTE_ADDR -> 128.151.67.228:8080 SIZE:600
#133 <1.738403sec> NET RECV: PID:13661 HOST_ADDR -> 128.151.67.29:41800 REMOTE_ADDR -> 128.151.67.228:8080 SIZE:568
#134 <1.738421sec> NET SEND: PID:13661 HOST_ADDR -> 128.151.67.29:41800 REMOTE_ADDR -> 128.151.67.228:8080 SIZE:363
#135 <1.752501sec> NET RECV: PID:13661 HOST_ADDR -> 128.151.67.29:41800 REMOTE_ADDR -> 128.151.67.228:8080 SIZE:12
```

Figure 4: Identifying blocking round trip communications based on an intercepted network system call trace. #132⟶#135 is counted as only one round trip interaction because #132, #133, and #134 occur very close to each other (less than a single network round-trip latency in the profiling environment).

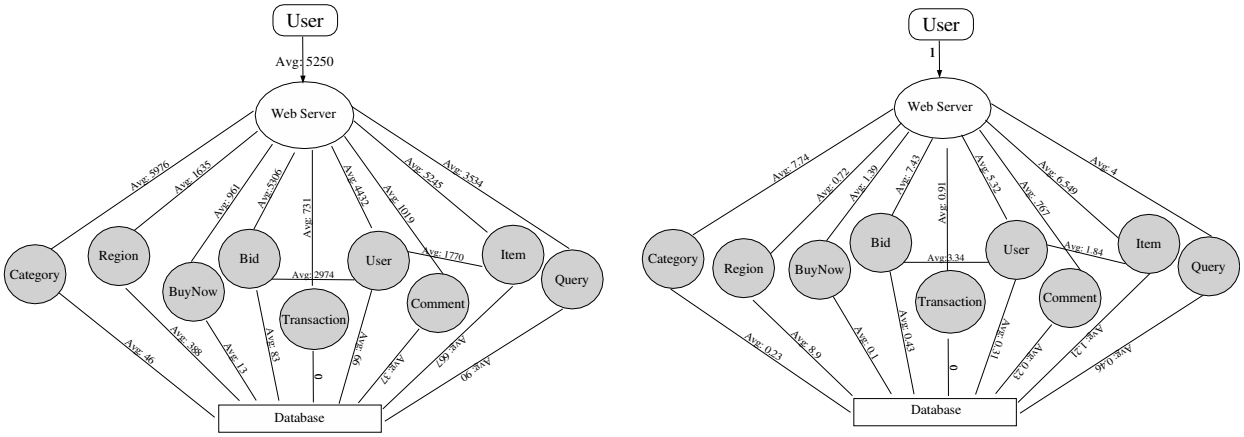


Figure 5: Inter-component communication profile for RUBiS. In the left figure, the label on each edge indicates the per-request mean of *data communication volume* in bytes. In the right figure, the label on each edge indicates the per-request mean of *the blocking round-trip communication count*.

This eliminates the interference from co-located components without the need of complex noise-reduction techniques [8]. However, the profiling of isolated components imposes a large demand on the profiling infrastructure (*i.e.*, the cluster size equals the number of components). With a smaller profiling infrastructure, it would require multiple profile runs that each measures some components or inter-component links. At a minimum, two cluster nodes are needed for per-component resource consumption profiling (one for the profiled component and the other for the rest) and three machines are required to measure inter-component communications. At these settings for an N-component service, it would take N profile runs for per-component measurement and $\frac{N \cdot (N-1)}{2}$ runs for inter-component communication profiling.

3 Performance Modeling

We present our performance model for cluster-based multi-component online services. The input of our model includes the application profile, workload properties, available resources in the hosting platform, as well as the component placement and replication strategy. Below we describe our throughput prediction (Section 3.1) and response time prediction (Section 3.2) schemes. We discuss several remaining issues about our performance model in Section 3.3.

3.1 Throughput Prediction

Our ability to project system throughput under each component placement strategy can be illustrated by the following three-step process:

1. We first derive the mapping between the input request rate $\lambda_{\text{workload}}$ and runtime resource demands for each component. The CPU, disk I/O bandwidth, and memory needs $\theta_{\text{cpu}}, \theta_{\text{disk}}, \theta_{\text{mem}}$ can be obtained with the knowledge of the component resource consumption profile (Section 2.1) and input workload characteristics. The remote invocation overhead (Section 2.2) is then added when the component in question interacts with other components that are placed on remote servers. The component network resource demand θ_{network} can be derived from the inter-component communication profile (Section 2.3) and the placement strategy. More specifically, it is the sum of communication data volume on all non-local inter-component links adjacent to the component in question.

2. Given a component placement strategy, we can determine the maximum input request rate that can saturate the CPU, disk I/O bandwidth, network I/O

bandwidth, or memory resources at each server. Let $\tau_{\text{CPU}}(s)$, $\tau_{\text{disk}}(s)$, $\tau_{\text{network}}(s)$, and $\tau_{\text{memory}}(s)$ denote such saturation rates at server s. When a component is replicated, its load is distributed to all server replicas. The exact load distribution is dependent on the policy employed by the component middleware. Many load distribution strategies have been proposed in previous studies [13, 25, 26, 40]. However, most of these have not been incorporated into component middleware systems in practice. Particularly, we are only able to employ round-robin load distribution with the JBoss application server, which evenly distributes the load among replicas. Our validation results in Section 4 are therefore based on the round-robin load distribution model.

3. The system reaches its maximum throughput as soon as one of the servers cannot handle any more load. Therefore, the system throughput can be estimated as the lowest saturation rate for all resource types at all servers:

$$\min_{\text{for all servers}} \{\tau_{\text{CPU}}(s), \tau_{\text{disk}}(s), \tau_{\text{network}}(s), \tau_{\text{memory}}(s)\}$$

3.2 Response Time Prediction

The service response time for a cluster-based online service includes two elements: 1) the request execution time and the queueing time caused by resource competition; and 2) network delay due to blocking inter-component communications.

Request Execution and Queuing The system-wide request execution and the queueing time is the sum of such delay at each cluster server. We use an M/G/1 queue to simplify our model of the average response time at each server. The M/G/1 queue assumes independent request arrivals where the inter-arrival times follow an exponential distribution. Previous studies [7, 11] found that Web request arrivals may not be independent because Web workloads contain many automatically triggered requests (*e.g.*, embedded Web objects) and requests within each user session may follow particular user behavioral patterns. We believe our simplification is justified because our model targets dynamic-content service requests that do not include automatically triggered embedded requests. Additionally, busy online services observe superimposed workloads from many independent user sessions and thus the request inter-dependencies with individual user sessions are not pronounced, particularly at high concurrencies.

The average response time at each server can be estimated as follows under the M/G/1 queuing model:

$$E[r] = E[e] + \frac{\rho E[e](1 + C_e^2)}{2(1 - \rho)}$$

where $E[e]$ is the average request execution time; ρ is the workload intensity (*i.e.*, the ratio of input request rate to the rate at which the server resource is completely exhausted); and C_e is the coefficient of variation (*i.e.*, the ratio of standard deviation to the sample mean) of the request execution time.

The average request execution time at each component is the resource needs per request at very low request rate (when there is no resource competition or queuing in the system). The average request execution time at each server $E[e]$ is the sum of such time for all hosted components. The workload intensity at each server ρ can be derived from the component resource needs and the set of components that are placed at the server in question. The coefficient of variation of the request execution time C_e can be determined with the knowledge of its distribution. For instance, $C_e = 1$ if the request execution time follows an exponential distribution. Without the knowledge of such distribution, our application profile maintains an histogram of execution time samples for each component and we then use these histograms to determine C_e for each server under a particular placement strategy.

Network Delay Our network delay prediction is based on the inter-component communication profile described in Section 2.3. From the profile, we can acquire the per-request communication data volume (denoted by ν_l) and round-trip blocking interaction count (denoted by κ_l) for each inter-component link l. Under a particular cluster environment, let γ_l and ω_l be the round-trip latency and bandwidth, respectively, of link l. Then we can model the per-request total network delay as:

$$\Gamma = \sum_{\text{for each non-local inter-component link } l} [\kappa_l \cdot \gamma_l + \frac{\nu_l}{\omega_l}]$$

3.3 Additional Modeling Issues

Cache pollution due to component co-location. When components are co-located, interleaved or concurrent execution of multiple components may cause processor cache pollution on the server, and thus affect the system performance. Since modern operating systems employ affinity-based scheduling and large CPU quanta (compared with the cache warm-up time), we do not find cache pollution to be a significant performance factor. However, processor-level multi-threading technologies such as the Intel Hyper-Threading [19] allow concurrent threads executing on a single processor and sharing level-1 processor cache. Cache pollution is more pronounced on these architectures and it might be necessary to model such cost and its impact on the overall system performance.

Replication consistency management. If replicated application states can be updated by user requests, mechanisms such as logging, undo, and redo may be required

to maintain consistency among replicated states. Previous studies have investigated replication consistency management for scalable online services [15, 29, 32, 39]. Consistency management consumes additional system resources which our current model does not consider. Since many component middleware systems in practice do not support replication consistency management, we believe this limitation of our current model should not severely restrict its applicability.

Cross-architecture performance modeling. A service hosting cluster may comprise servers with multiple types of processor architectures or memory sizes. Our current approach requires application profiling on each of the architectures present in the cluster. Such profiling is time consuming and it would be desirable to separate the performance impact of application characteristics from that of server properties. This would allow independent application profiling and server calibration, and thus significantly save the profiling overhead for server clusters containing heterogeneous architectures. Several recent studies [22, 33] have explored this issue in the context of scientific computing applications and their results may be leveraged for our purpose.

4 Model Validation

We perform measurements to validate the accuracy of our throughput and response time prediction models. An additional objective of our measurements is to identify the contributions of various factors on our model accuracy. We are particularly interested in the effects of the remote invocation overhead modeling and the network delay modeling.

Our validation measurements are based on the RUBiS and StockOnline applications described in Section 2.1. The application EJB components are hosted on JBoss 3.2.3 application servers with embedded Tomcat 5.0 servlet containers. The database servers run MySQL 4.0. Although MySQL 4.0 supports master/slave replication, the JBoss application server cannot be configured to access replicated databases. Therefore we do not replicate the database server in our experiments.

All measurements are conducted on a 20-node heterogeneous Linux cluster connected by a 1 Gbps Ethernet switch. The roundtrip latency (UDP or TCP without connection setup) between two cluster nodes takes around 150 μs. The cluster nodes have three types of hardware configurations. Each type-1 server is equipped with a 2.66 GHz Pentium 4 processor and 512 MB memory. Each type-2 server is equipped with a single 2.00 GHz Xeon processor and 512 MB memory. Each type-3 server is equipped with two 1.26 GHz Pentium III processors and 1 GB memory. All application data is hosted on two local 10 KRPM SCSI drives at each server.

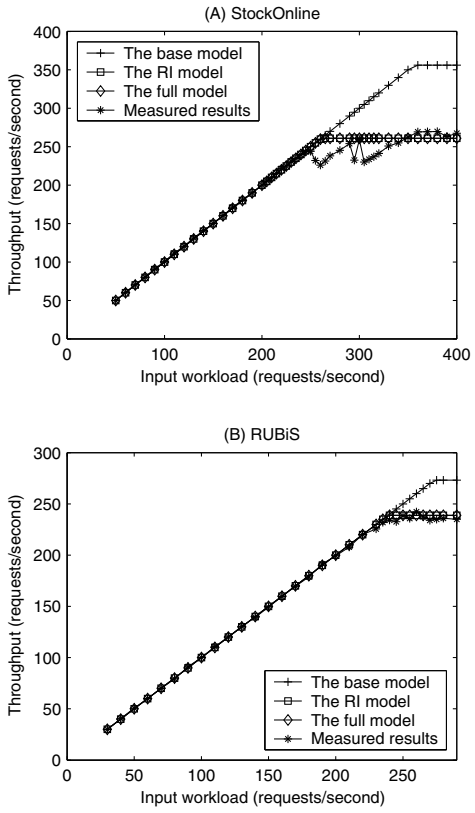


Figure 6: Validation results on system throughput.

The performance of a cluster-based online service is affected by many factors, including the cluster size, the mix of input request types, the heterogeneity of the hosting servers, as well as the placement and replication strategy. Our approach is to first provide detailed results on the model accuracy at a typical setting (Section 4.1) and then explicitly evaluate the impact of various factors (Section 4.2). We summarize our validation results in Section 4.3.

4.1 Model Accuracy at a Typical Setting

The measurement results in this section are based on a 12-machine service cluster (four are type-1 and the other eight are type-2). For each application, we employ an input request mix with 10% read-write requests and 90% read-only requests. The placement strategy for each application we use (shown in Table 3) is the one with highest modeled throughput out of 100 random chosen candidate strategies. We compare the measured performance with three variations of our performance model:

#1. *The base model:* The performance model that does not consider the overhead of remote component invocation and network delays.

Servers	Pentium4	Pentium4	Pentium4	Pentium4	Xeon	Xeon	Xeon	Xeon	Xeon	Xeon	Xeon	Xeon
Stock Online	Account Item Holding StockTX Broker	Account Item Holding StockTX Broker	Account Holding StockTX Broker	Account Holding StockTX Broker	Account Holding StockTX Broker WS	Account Holding StockTX Broker WS	Account Holding StockTX Broker WS	Account Holding StockTX Broker WS	Account Holding StockTX Broker WS	Account StockTX Broker WS	Account Holding Broker WS	Account Holding StockTX Broker DB
RUBiS	Bid Query	Item	User	Region BuyNow	BuyNow Trans. WS	Region BuyNow WS	BuyNow WS	WS	WS	Region Trans. Query	BuyNow Comment DB	Category

Table 3: Component placement strategies (on a 12-node cluster) used for measurements in Section 4.1.

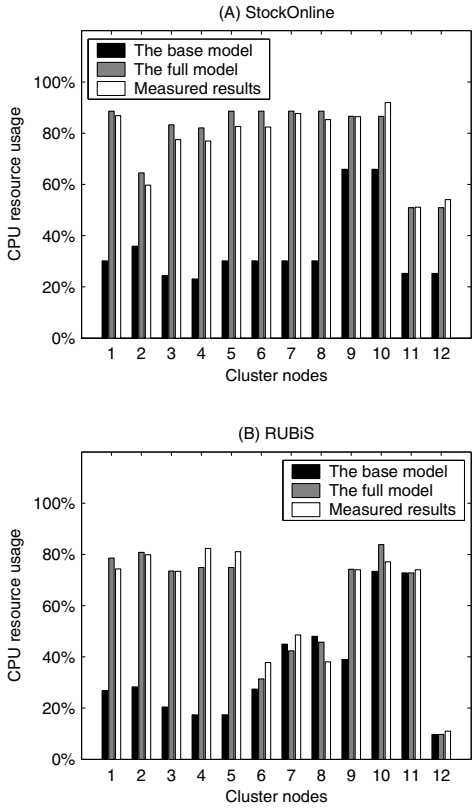


Figure 7: Validation results on per-node CPU resource usage (at the input workload rate of 230 requests/second).

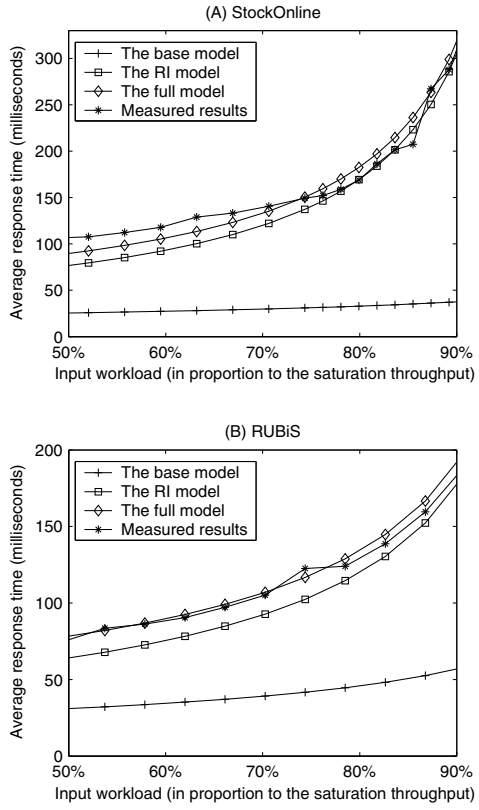


Figure 8: Validation results on service response time.

#2. *The RI model:* The performance model that considers remote invocation overhead but not network delays.
#3. *The full model:* The performance model that considers both.

Figure 6 shows validation results on the overall system throughput for StockOnline and RUBiS. We measure the rate of successfully completed requests at different input request rate. In our experiments, a request is counted as successful only if it returns within 10 seconds. Results show that our performance model can accurately predict system throughput. The error for RUBiS is negligible while the error for StockOnline is less than 13%. This error is mostly attributed to instable results (due to timeouts) when the system approaches the saturation point. There is no difference between the prediction of the RI model and that of the full model. This is expected since the network delay modeling does not affect the component resource needs and subsequently the system throughput. Comparing between the full model and the base model, we find that the modeling of remote invocation overhead has a large impact on the prediction accuracy. It improves the accuracy by 36% and 14% for StockOnline and RUBiS respectively.

Since the system throughput in our model is derived from resource usage at each server, we further examine the accuracy of per-node resource usage prediction. Figure 7 shows validation results on the CPU resource usage at the input workload rate of 230 requests/second (around 90% workload intensity for both applications). We do not show the RI model since its resource usage prediction is the same as the full model. Comparing between the full model and the base model, we find that the

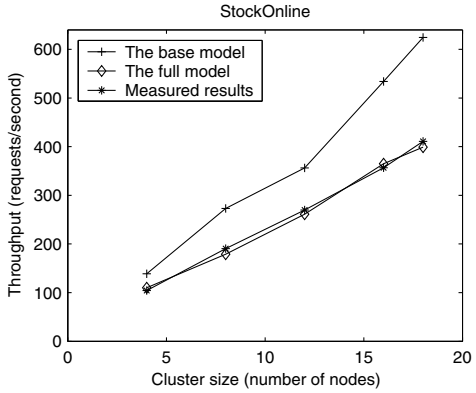


Figure 9: Validation results on system saturation throughput at various cluster sizes.

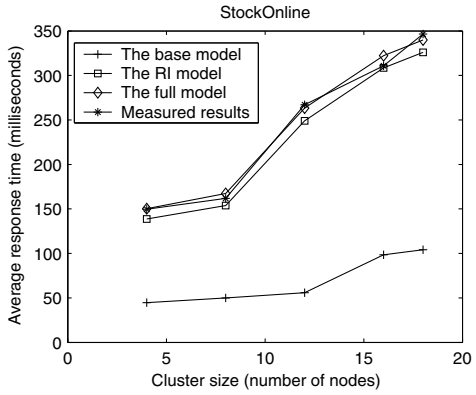


Figure 10: Validation results on service response time at various cluster sizes. The response time is measured at the input workload that is around 85% of the saturation throughput.

remote invocation overhead can be very significant on some of the servers. The failure of accounting it results in poor performance prediction of the base model.

Figure 8 shows validation results on the average service response time for StockOnline and RUBiS. For each application, we show the average response time when the input workload is between 50% and 90% of the *saturation throughput*, defined as the highest successful request completion rate achieved at any input request rate. Results show that our performance model can predict the average response time with less than 14% error for the two applications. The base model prediction is very poor due to its low resource usage estimation. Comparing between the full model and the RI model, we find that the network delay modeling accounts for an improved accuracy of 9% and 18% for StockOnline and RUBiS respectively.

4.2 Impact of Factors

We examine the impact of various factors on the accuracy of our performance model. When we vary one factor, we keep other factors unchanged from settings in

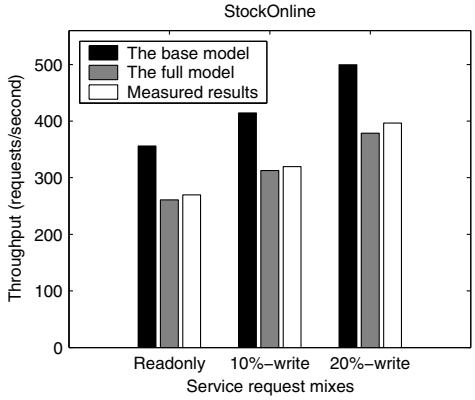


Figure 11: Validation results on system saturation throughput at various service request mixes.

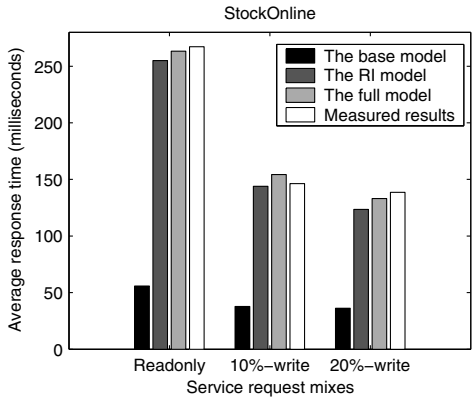


Figure 12: Validation results on service response time at various service request mixes. The response time is measured at the input workload that is around 85% of the saturation throughput.

Section 4.1.

Impact of cluster sizes We study the model accuracy at different cluster sizes. Figure 9 shows the throughput prediction for the StockOnline application at service clusters of 4, 8, 12, 16, and 18 machines. At each cluster size, we pick a high-throughput placement strategy out of 100 random chosen candidate placements. Figure 10 shows the validation results on the average response time. The response time is measured at the input workload that is around 85% of the saturation throughput. Results demonstrate that the accuracy of our model is not affected by the cluster size. The relative accuracies among different models are also consistent across all cluster sizes.

Impact of request mixes Figure 11 shows the throughput prediction for StockOnline at input request mixes with no writes, 10% read-write request sequences, and 20% read-write request sequences. Figure 12 shows the validation results on the average response time. Results

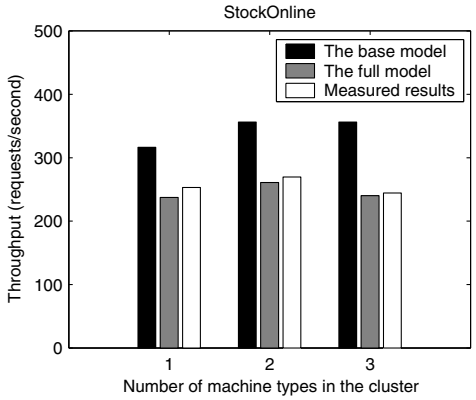


Figure 13: Validation results on system saturation throughput at various cluster settings.

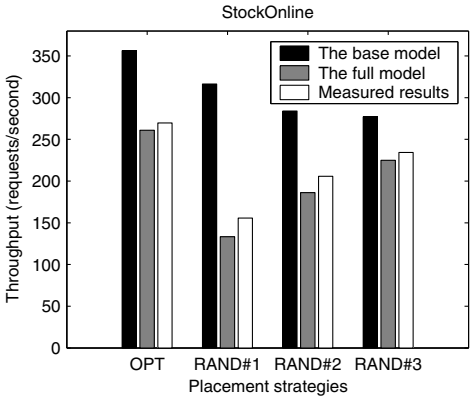


Figure 15: Validation results on system saturation throughput at various placement strategies.

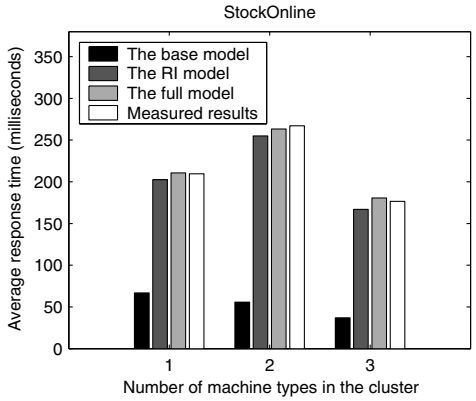


Figure 14: Validation results on service response time at various cluster settings. The response time is measured at the input workload that is around 85% of the saturation throughput.

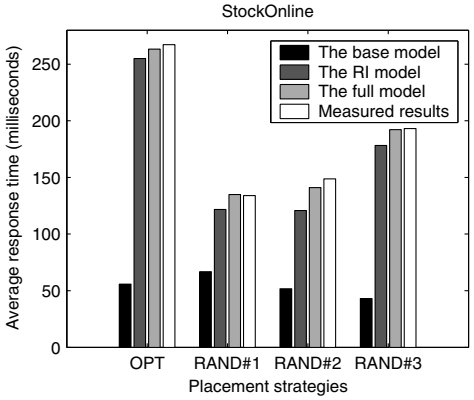


Figure 16: Validation results on service response time at various placement strategies. The response time is measured at the input workload that is around 85% of the saturation throughput.

demonstrate that the accuracy of our model is not affected by different types of input requests.

Impact of heterogeneous machine architectures Figure 13 shows the throughput prediction for StockOnline at service clusters with one, two, and three types of machines. All configurations have twelve machines in total. Figure 14 illustrates the validation results on the average response time. Results show that the accuracy of our model is not affected by heterogeneous machine architectures.

Impact of placement and replication strategies Finally, we study the model accuracy at different component placement and replication strategies. Figure 15 shows the throughput prediction for StockOnline with the throughput-optimized placement and three random chosen placement strategies. Methods for finding high-performance placement strategies will be discussed in Section 5.1. Figure 16 shows the validation results on the average response time. Small prediction errors are observed at all validation cases.

4.3 Summary of Validation Results

- Our model can predict the performance of the two J2EE applications with high accuracy (less than 13% error for throughput and less than 14% error for the average response time).
- The remote invocation overhead is very important for the accuracy of our performance model. Without considering it, the throughput prediction error can be up to 36% while the prediction for the average response time can be much worse depending on the workload intensity.
- Network delay can affect the prediction accuracy for the average response time by up to 18%. It does not affect the throughput prediction. However, the impact of network delay may increase with networks of lower bandwidth or longer latency.
- The validation results are not significantly affected by factors such as the cluster size, the mix of input request types, the heterogeneity of the hosting

Servers	Pentium4	Pentium4	Pentium4	Pentium4	Xeon	Xeon	Xeon	Xeon	Xeon	Xeon	Xeon	Xeon
Simulate annealing OPT	Holding StockTX Broker DB	Item StockTX Broker	Item StockTX Broker	Item Holding Broker	Holding Broker WS	Holding Broker WS	Holding Broker WS	Account Broker WS	Account Broker WS	Account Broker WS	Account Broker WS	Account Broker WS
Random sampling OPT	Item StockTX Broker WS	Item Broker DB	Account Item Holding StockTX Broker	Holding StockTX Broker	Account Item StockTX Broker WS	Account Item StockTX Broker WS	Account Item StockTX Broker WS	Account Item StockTX Broker WS	Account Item StockTX Broker WS	Account Item StockTX Broker WS	Account Item StockTX Broker WS	Account Item StockTX Broker WS
Low replication	Item Broker	Item Broker	WS	WS	Account Broker	Holding Broker	StockTX Broker	Broker	Broker WS	Broker WS	Broker WS	DB

Table 4: Component placement strategies (on a 12-node cluster) used for measurements in Section 5.1. The "all replication" strategy is not shown.

servers, and the placement strategy.

5 Model-based System Management

We explore how our performance model can be used to assist system management functions for multi-component online services. A key advantage of model-based management is its ability to quickly explore the performance tradeoff among a large number of system configuration alternatives without high-overhead measurements. Additionally, it can project system performance at hypothetical settings.

5.1 High-performance Component Placement

Our objective is to discover a component placement and replication strategy that achieves high performance on both throughput and service response time. More specifically, our target strategy should be able to support a large input request rate while still maintaining an average response time below a specified threshold. Our model proposed in this paper can predict the performance with any given component placement strategy. However, the search space of all possible placement strategies is too large for exhaustive check. Under such a context, we employ optimization by simulated annealing [21, 24]. Simulated annealing is a random sampling-based optimization algorithm that gradually reduces the sampling scope following an "annealing schedule".

We evaluate the effectiveness of our approach on a 12-node cluster (the same as in Section 4.1) using the StockOnline application. We set the response time threshold of our optimization at 1 second. The number of samples examined by our simulated annealing algorithm is in the order of 10,000 and the algorithm takes about 12 seconds to complete on a 2.00 GHz Xeon processor. For comparison, we also consider a random sampling optimization which selects the best placement out of 10,000 randomly chosen placement strategies. Note that both of these approaches rely on our performance model.

For additional comparison, we introduce two placement strategies based on "common sense", *i.e.*, without the guidance of our performance model. In the first strategy, we replicate all components (except the database) on all nodes. This is the suggested placement strategy in the JBoss application server documentation. High replication may introduce some overhead, such as the component maintenance overhead at each replica that is not directly associated with serving user requests. In the other strategy, we attempt to minimize the amount of replication while still maintaining balanced load. We call this strategy *low replication*. Table 4 lists the three placement strategies except all replication.

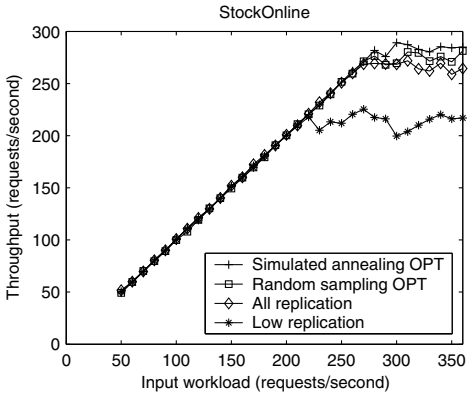


Figure 17: System throughput under various placement strategies.

Figure 17 illustrates the measured system throughput under the above four placement strategies. We find that the simulated annealing optimization is slightly better than the random sampling approach. It outperforms all replication and low replication by 7% and 31% respectively. Figure 18 shows the measured average response time at different input workload rates. The response time for low replication rises dramatically when approaching 220 requests/second because it has a much lower saturation throughput than the other strategies. Compared with random sampling and all replication, the simulated annealing optimization achieves 22% and 26% lower response time, respectively, at the input workload rate of 250 requests/second.

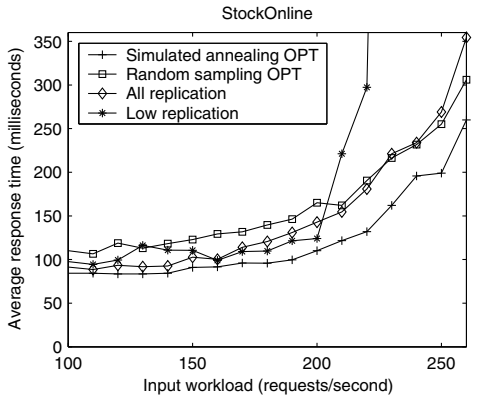


Figure 18: Average service response time under various placement strategies.

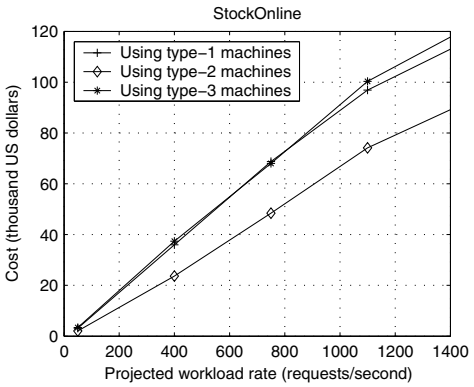


Figure 20: Cost-effectiveness of various machine types.

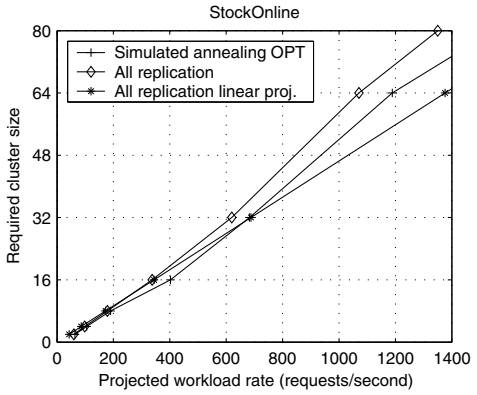


Figure 19: Capacity planning using various models.

5.2 Capacity Planning

The ability of predicting future resource needs at forecast workload levels allows an online service provider to acquire resources in an efficient fashion. Figure 19 presents our capacity planning results for StockOnline on simulated annealing-optimized placement and all replication placement. The base platform for capacity planning is a 12-node cluster (four type-1 nodes and eight type-2 nodes). Our performance model is used to project resource needs at workload levels that could not be supported in the base platform. We assume only type-2 nodes will be added in the future. Previous work [23] has suggested linear projection-based capacity planning where future resource requirement scales linearly with the forecast workload level. For the comparison purpose, we also show the result of linear projection. The base performance for linear projection is that of all replication on the 12-node cluster.

Results in Figure 19 illustrate that our optimized strategy consistently saves resources compared with all replication. The saving is at least 11% for projected workload of 1000 requests/second or higher. Comparing between the modeled all replication and linearly projected all replication, we find that the linear projection significantly underestimates resource needs (by about 28%) at high workload levels. This is partially due to heterogeneity in the machines being employed. More specifically, four of the original nodes are type-1, which are slightly more powerful than the type-2 machines that are expected to be added. Additionally, linear projection fails to consider the increased likelihood of remote invocations at larger clusters. In comparison, our performance model addresses these issues and provides more accurate capacity planning.

5.3 Cost-effectiveness Analysis

We provide a model-based cost-effectiveness analysis. In our evaluation, we plan to choose one of the three types of machines (described in the beginning of Section 4) to expand the cluster for future workloads. We examine the StockOnline application in this evaluation. Figure 20 shows the estimated cost when each of the three types of machines is employed for expansion. We acquire the pricing for the three machine types from `www.epinions.com` and they are $1563, $1030, and $1700 respectively. Although type-2 nodes are less powerful than the other types, it is the most cost-effective choice for supporting the StockOnline application. The saving is at least 20% for projected workload of 1000 requests/second or higher. Such a cost-effectiveness analysis would not be possible without an accurate prediction of application performance at hypothetical settings.

6 Related Work

Previous studies have addressed application resource consumption profiling. Urgaonkar *et al.* use resource usage profiling to guide application placement in shared hosting platforms [36]. Amza *et al.* provide bottleneck resource analysis for several dynamic-content online ser-

vice benchmarks [3]. Doyle *et al.* model the service response time reduction with increased memory cache size for static-content Web servers [12]. The Magpie tool chain actually extracts per-request execution control flow through online profiling [8]. The main contribution of our profiling work is that we identify a comprehensive set of application characteristics that can be employed to predict the performance of multi-component online services with high accuracy.

A very recent work by Urgaonkar *et al.* [35] models a multi-tier Internet service as a network of queues. Their view of service tiers is not as fine-grained as application components in our model. Additionally, service tiers are organized in a chain-like architecture while application components can interact with each other in more complex fashions. As a result, their model cannot be used to directly guide component-level system management such as distributed component placement. On the other hand, our approach uses a simple M/G/1 queue to model service delay at each server while their model more accurately captures the dependencies of the request arrival processes at different service tiers.

Recent studies have proposed the concept of component-oriented performance modeling [17, 37]. They mainly focus on the design of performance characterization language for software components and the way to assemble component-level models into whole-application performance model. They do not describe how component performance characteristics can be acquired in practice. In particular, our study finds that the failure of accounting the remote invocation overhead can significantly affect the model accuracy.

Distributed component placement has been examined in a number of prior studies. Coign [18] examines the optimization problem of minimizing communication time for two-machine client-server applications. ABACUS [2] focuses on the placement of I/O-specific functions for cluster-based data-intensive applications. Ivan *et al.* examine the automatic deployment of component-based software over the Internet subjected to throughput requirements [20]. Most of these studies heuristically optimize the component placement toward a performance objective. In comparison, our model-based approach allows the flexibility to optimize for complex objectives (*e.g.*, a combination of throughput and service response time) and it also provides an estimation on the maximum achievable performance.

7 Conclusion and Future Work

This paper presents a profile-driven performance model for cluster-based multi-component online services. We construct application profiles characterizing component resource needs and inter-component communication patterns using transparent operating system instrumentation. Given a component placement and replication strategy, our model can predict system throughput and the average service response time with high accuracy. We demonstrate how this performance model can be employed to assist optimized component placement, capacity planning, and cost-effectiveness analysis.

In addition to supporting static component placement, our model may also be used to guide dynamic runtime component migration for achieving better performance. Component migration requires up-to-date knowledge of runtime dynamic workload characteristics. It also desires a migration mechanism that does not significantly disrupt ongoing service processing. Additionally, runtime component migration must consider system stability, especially when migration decisions are made in a decentralized fashion. We plan to investigate these issues in the future.

Project Website More information about this work, including publications and releases of related tools and documentations can be found on our project website: **www.cs.rochester.edu/u/stewart/component.html**.

Acknowledgments We would like to thank Peter DeRosa, Sandhya Dwarkadas, Michael L. Scott, Jian Yin, Ming Zhong, the anonymous referees, and our shepherd Dina Katabi for helpful discussions and valuable comments. We are also indebted to Liudvikas Bukys and the rest of the URCS lab staff for maintaining the experimental platform used in this study.

References

[1] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: An Automated Resource Provisioning Tool for Large-Scale Storage Systems. *ACM Trans. on Computer Systems*, 19(4):483–418, November 2001.

[2] K. Amiri, D. Petrou, G. Ganger, and G. Gibson. Dynamic Function Placement for Data-Intensive Cluster Computing. In *Proc. of the USENIX Annual Technical Conf.*, San Diego, CA, June 2000.

[3] C. Amza, A. Chanda, A. L. Cox, S. Elnikety, R. Gil, K. Rajamani, W. Zwaenepoel, E. Cecchet, and J. Marguerite. Specification and Implementation of Dynamic Web Site Benchmarks. In *Proc. of the 5th IEEE Workshop on Workload Characterization*, Austin, TX, November 2002.

[4] E. Anderson. Simple Table-based Modeling of Storage Devices. Technical Report HPL-SSP-2001-04, HP Laboratories, July 2001.

[5] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger. Oceano – SLA Based Management of A Computing Utility. In *Proc. of the 7th Int'l Symp. on Integrated Network Management*, Seattle, WA, May 2001.

[6] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster Reserve: A Mechanism for Resource Management in Cluster-based Network Servers. In *Proc. of the ACM SIGMETRICS*, pages 90–101, Santa Clara, CA, June 2000.

[7] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proc. of the ACM SIGMETRICS*, pages 151–160, Madison, WI, June 1998.

[8] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modeling. In *Proc. of the 6th USENIX OSDI*, pages 259–272, San Francisco, CA, December 2004.

[9] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and Scalability of EJB Applications. In *Proc. of the 17th ACM Conf. on Object-oriented Programming, Systems, Languages, and Applications*, pages 246–261, Seattle, WA, November 2002.

[10] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyl. Managing Energy and Server Resources in Hosting Centers. In *Proc. of the 18th ACM SOSP*, pages 103–116, Banff, Canada, October 2001.

[11] S. Deng. Empirical Model of WWW Document Arrivals at Access Link. In *Proc. of the IEEE Conf. on Communications*, pages 1797–1802, Dallas, TX, June 1996.

[12] R. P. Doyle, J. S. Chase, O. M. Asad, W. Jin, and A. M. Vahdat. Model-based Resource Provisioning in a Web Service Utility. In *Proc. of the 4th USENIX Symp. on Internet Technologies and Systems*, Seattle, WA, March 2003.

[13] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Trans. on Software Engineering*, 12(5):662–675, May 1986.

[14] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *Proc. of the 16th ACM SOSP*, pages 78–91, Saint Malo, France, October 1997.

[15] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proc. of the 4th USENIX OSDI*, San Diego, CA, October 2000.

[16] S. D. Gribble, M. Welsh, E. A. Brewer, and D. Culler. The MultiSpace: An Evolutionary Platform for Infrastructural Services. In *Proc. of the USENIX Annual Technical Conf.*, Monterey, CA, June 1999.

[17] S. A. Hissam, G. A. Moreno, J. A. Stafford, and K. C. Wallnau. Packaging Predictable Assembly. In *Proc. of the First IFIP/ACM Conf. on Component Deployment*, Berlin, Germany, June 2002.

[18] G. C. Hunt and M. L. Scott. The Coign Automatic Distributed Partitioning System. In *Proc. of the Third USENIX OSDI*, New Orleans, LA, February 1999.

[19] Hyper-Threading Technology. http://www.intel.com /technology/hyperthread.

[20] A.-A. Ivan, J. Harman, M. Allen, and V. Karamcheti. Partitionable Services: A Framework for Seamlessly Adapting Distributed Applications to Heterogeneous Environments. In *Proc. of the 11th IEEE Symp. on High Performance Distributed Computing*, pages 103–112, Edinburgh, Scotland, July 2002.

[21] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, May 1983.

[22] G. Marin and J. Mellor-Crummey. Cross-Architecture Performance Predictions for Scientific Applications Using Parameterized Models. In *Proc. of the ACM SIGMETRICS*, pages 2–13, New York, NY, June 2004.

[23] D. A. Menascé and V. A. F. Almeida. *Capacity Planning for Web Performance: Metrics, Models, and Methods*. Prentice Hall, 1998.

[24] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of State Calculations by Fast Computing Machines. *J. Chem. Phys.*, 21(6):1087–1092, 1953.

[25] M. Mitzenmacher. On the Analysis of Randomized Load Balancing Schemes. In *Proc. of the 9th ACM SPAA*, pages 292–301, Newport, RI, June 1997.

[26] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proc. of the 8th ASPLOS*, pages 205–216, San Jose, CA, October 1998.

[27] D. Petriu and M. Woodside. Analysing Software Requirements Specifications for Performance. In *Proc. of the Third Int'l Workshop on Software and Performance*, Rome, Italy, July 2002.

[28] RUBiS: Rice University Bidding System. http://rubis .objectweb.org.

[29] Y. Saito, B. N. Bershad, and H. M. Levy. Manageability, Availability, and Performance in Porcupine: a Highly Scalable, Cluster-based Mail Service. *ACM Trans. on Computer Systems*, 18(3):298–332, August 2001.

[30] G. A. Seber. *Nonlinear Regression Analysis*. Wiley, New York, 1989.

[31] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated Resource Management for Cluster-based Internet Services. In *Proc. of the 5th USENIX OSDI*, pages 225–238, Boston, MA, December 2002.

[32] K. Shen, T. Yang, and L. Chu. Clustering Support and Replication Management for Scalable Network Services. *IEEE Trans. on Parallel and Distributed Systems - Special Issue on Middleware*, 14(11):1168–1179, November 2003.

[33] A. Snavely, L. Carrington, and N. Wolter. Modeling Application Performance by Convolving Machine Signatures with Application Profiles. In *Proc. of the 4th IEEE Workshop on Workload Characterization*, Austin, TX, December 2001.

[34] The StockOnline Benchmark. http://forge.objectweb.org /projects/stock-online.

[35] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An Analytical Model for Multi-tier Internet Services and Its Applications. In *Proc. of the ACM SIGMETRICS (to appear)*, Banff, Canada, June 2005.

[36] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. In *Proc. of the 5th USENIX OSDI*, pages 239–254, Boston, MA, December 2002.

[37] X. Wu and M. Woodside. Performance Modeling from Software Components. In *Proc. of the 4th Int'l Workshop on Software and Performance*, pages 290–301, Redwood City, CA, January 2004.

[38] K. Yaghmour and M. R. Dagenais. Measuring and Characterizing System Behavior Using Kernel-Level Event Logging. In *Proc. of the USENIX Annual Technical Conf.*, San Diego, CA, June 2000.

[39] H. Yu and A. Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *Proc. of the 4th USENIX OSDI*, San Diego, CA, October 2000.

[40] S. Zhou. A Trace-Driven Simulation Study of Dynamic Load Balancing. *IEEE Trans. on Software Engineering*, 14(9):1327–1341, September 1988.

Debunking some myths about structured and unstructured overlays

Miguel Castro *Manuel Costa* *Antony Rowstron*

Microsoft Research, 7 J J Thomson Avenue, Cambridge, UK

Abstract

We present a comparison of structured and unstructured overlays that decouples overlay topology maintenance from query mechanism. Structured overlays provide efficient support for simple exact-match queries but they constrain overlay topology to achieve this. Unstructured overlays do not constrain overlay topology or query complexity because they use flooding or random walks to discover data. It is commonly believed that structured overlays are more expensive to maintain, that their topology constraints make it harder to exploit heterogeneity, and that they cannot support complex queries efficiently. We performed a detailed comparison study using simulations driven by real-world traces that debunks these widespread myths. We describe techniques that exploit structural constraints to achieve low maintenance overhead and we present a modified neighbour selection algorithm that can exploit heterogeneity effectively. We also describe techniques to perform floods and random walks on structured topologies. These techniques exploit structural constraints to support complex queries with better performance than unstructured overlays.

1 Introduction

There has been much interest in peer-to-peer data sharing applications. They are used by millions of users and they represent a large fraction of the traffic in the Internet [31]. These applications are built on top of large-scale network overlays that provide mechanisms to discover data stored by overlay nodes. There is an ongoing debate in the research community on the relative merits of two types of overlays: unstructured and structured. This paper presents a comparison study of unstructured and structured overlays that contributes to this debate by debunking some widespread myths.

Unstructured overlays, for example Gnutella [1], organize nodes into a random graph topology and use floods or random walks to discover data stored by overlay nodes. Each node visited during a flood or random walk evaluates the query locally on the data items that it stores. This approach supports arbitrarily complex queries and it does not impose any constraints on the overlay topology or on data placement, for example, each node can choose any other node to be its neighbour in the overlay and it can store the data it owns. There has been a large amount of work on improving unstructured overlays, for example [10, 13, 24].

Structured overlays, like Tapestry [35], CAN [25], Chord [32] and Pastry [29], were developed to improve the performance of data discovery. They impose constraints both on the topology of the overlay and on data placement to enable efficient discovery of data. Each data item is identified by a key and nodes are organized into a structured graph topology that maps each key to a responsible node. The data or a pointer to the data is stored at the node responsible for its key. These constraints provide efficient support for exact-match queries; they enable discovery of a data item given its key in typically only $O(logN)$ hops with only $O(logN)$ neighbours per node. It is possible to support more complex queries by building indices on top of structured overlays but current solutions perform worse than unstructured overlays [20].

It is commonly believed that structured overlays are more expensive to maintain in the presence of churn, that their topology constraints remove the flexibility necessary to exploit heterogeneity, and that they cannot support complex queries efficiently (see for example, [10]). This paper presents a detailed comparison of structured and unstructured overlays that contradicts these myths.

We explore the design space by decoupling overlay topology maintenance from query mechanisms.

- We evaluate a technique that exploits structure to reduce maintenance overhead. It eliminates redundant failure detection probes by using structure to partition failure detection responsibility and to locate nodes that need to be informed about failures

and new node arrivals. We show that this technique can achieve robustness to high rates of churn with overhead lower than unstructured overlays.

- We describe how to exploit heterogeneity by modifying any proximity neighbour selection algorithm [8, 35, 16] to adapt the topology such that the indegree of nodes matches their capacity.

- We introduce techniques to support complex queries efficiently on structured topologies without constraints on data placement. These techniques perform floods or random walks on structured topologies but exploit structural constraints to ensure that nodes are visited only once during a query, the number of visited nodes is controlled accurately, and the average capacity of nodes visited during a query is increased to better exploit heterogeneity. Additionally, they remove the need to maintain both a structured and an unstructured overlay to implement hybrid search strategies [22].

The paper presents results of detailed comparisons between several representative structured and unstructured overlay topology maintenance algorithms. These results were obtained using simulations driven by real-world traces of node arrivals and departures in the Gnutella file sharing application [30]. The results show that our techniques enable structured overlays to cope with high rates of churn and exploit heterogeneity effectively with a maintenance overhead comparable to that achieved by state-of-the-art unstructured overlays.

We also compared the performance of data discovery using several representative unstructured overlays and using our techniques to perform floods and random walks on structured overlays. We used a real trace of content distribution across nodes in the eDonkey peer-to-peer file sharing application [12] to drive the simulations. The results show that our techniques can discover data more often, faster, or with lower overhead.

The additional functionality provided by structured overlays has proven important to achieve scalability and efficiency in a wide range of applications. Structured overlays can emulate the functionality of unstructured overlays with comparable or even better performance.

In Section 2, we describe and compare structured and unstructured topology maintenance protocols assuming a homogeneous setting. Section 3 extends the structured topology maintenance protocol to exploit heterogeneity in peers' resources and compares this with unstructured topology maintenance protocols which exploit heterogeneity. Section 4 compares the performance of content discovery using random walks and flooding on both structured and unstructured topologies, and Section 5 presents our conclusions.

2 Topology maintenance with churn

Measurement studies of deployed peer-to-peer overlays have observed a high rate of churn [4, 17, 30]; nodes join and leave these overlays constantly. Therefore, peer-to-peer overlays should be able to cope with a high rate of churn.

Can unstructured overlays cope with churn better than structured overlays?

Each node maintains a set of neighbours to form an overlay. Structured overlays impose constraints on the overlay topology; nodes have identifiers and two nodes can be neighbours only if their identifiers satisfy certain constraints. Unstructured overlays do not impose constraints on neighbours. Both types of overlay can improve robustness to churn at the expense of increased maintenance overhead by increasing the number of neighbours per node and probing them more frequently to detect and replace failed neighbours.

It is believed that maintaining a structured overlay in the presence of churn is more expensive than maintaining an unstructured overlay because of the constraints on neighbour selection. This section shows that this is not necessarily the case. It is possible to use structure to achieve better robustness with lower maintenance overhead in a structured overlay.

Structured overlays also impose constraints on data placement that can result in high overhead under churn for some applications [5]. We study structured overlays without these constraints to keep the evaluation independent of any particular application. Data placement constraints do not result in significant overhead in several applications (for example, content distribution [9] and Web caching [19]) and the search technique in Section 4 does not constrain data placement at all.

This section describes the implementation of structured and unstructured overlay maintenance protocols in an homogeneous setting and compares their performance. The next section explains how to exploit heterogeneity.

2.1 Unstructured overlays

We implemented an unstructured overlay maintenance protocol based on the specification of Gnutella version 0.4 [15] but we added many optimizations to the protocol to ensure a fair comparison.

Gnutella 0.4 organizes overlay nodes into a random graph. Each node in the overlay maintains a neighbour table with the network addresses of its neighbours in the overlay. The neighbour tables are symmetric; if node x has node y in its neighbour table then node y has node x in its neighbour table. There is an upper and lower bound on the number of entries in each node's neighbour table.

A joining node uses a random walk starting from a *bootstrap* node, which is randomly chosen from the set of nodes already in the overlay, to find other nodes to fill its neighbour table. It sends the bootstrap node a *neighbour discovery* message with a counter that is initialized to the number of nodes required to fill its neighbour table. Upon receiving a discovery message, a node checks whether it has less neighbours than the upper bound. If this is the case, the node sends a message to the joining node inviting it to become a neighbour and decrements the counter in the neighbour discovery message. In either case, the neighbour discovery message is forwarded to a randomly chosen neighbour if the counter is still greater than zero. To increase resilience to node and network failures, all neighbour discovery messages are acknowledged. If a node does not receive an acknowledgement within a timeout, it selects another neighbour at random and forwards the neighbour discovery message to that neighbour.

In addition to joins, nodes need to detect failures and replace faulty neighbours. Every t seconds each node sends an *I'm alive* message to every node in its neighbour table. Since all nodes do the same and neighbour tables are symmetric, each node should receive a message from each neighbour in each t second period. If a node does not receive a message from a neighbour, it explicitly probes them and if no reply is received the node is assumed to be faulty. We used $t = 30$ seconds in this paper. Nodes maintain a cache of other nodes that they use to replace failed neighbours. If the cache is empty, they obtain new neighbours by sending a neighbour discovery message to a randomly chosen neighbour. All messages sent between the nodes are used to replace explicit *I'm alive* messages.

Simulation results show that this protocol leads to poor query performance because the neighbour table of a joining node and those of its neighbours are likely to share a significant fraction of nodes. This reduces the effectiveness of floods and random walks to discover data. We overcome this problem by forwarding the neighbour discovery message over a number of random hops after each neighbour invitation is sent. We add a hop counter to discovery messages that is set to R by every node that replies with a neighbour invitation. Nodes decrement the hop counter when they forward a discovery message and they only consider sending a neighbour invitation when the counter is less than or equal to zero. We used $R = 5$ in this paper as, from experimental evaluation, this provided good query performance with small increase in maintenance overheads.

We use unbiased random walks because we found that biasing the random walk to nodes with low degree reduces overhead but results in poor query performance. We also experimented with flooding of discovery messages (as specified in the Gnutella 0.4 protocol) but this results in additional overhead without improved robustness or query performance.

2.2 Structured overlays

There are several structured overlay maintenance protocols. We chose an implementation of Pastry [29] called MS Pastry [6] because it has good performance under churn and has an efficient implementation of proximity neighbour selection [8]. We modified it to exploit heterogeneity (as described in the next section). Studies have shown that other structured overlay maintenance protocols[21, 28] also perform well under churn.

Structured overlays map keys to overlay nodes. Overlay nodes are assigned *nodeIds* selected from a large identifier space and application objects are identified by keys selected from the same identifier space. Pastry selects nodeIds and keys uniformly at random from the set of 128-bit unsigned integers and it maps a key k to the node whose identifier is numerically closest to k modulo 2^{128}. This node is called the key's root. Given a message and a destination key, Pastry routes the message to the key's root node. Each node maintains a routing table and a leaf set to route messages.

NodeIds and keys are interpreted as a sequence of digits in base 2^b. We use $b = 1$ in this paper to minimizes the maintenance overhead. The routing table is a matrix with $128/b$ rows and 2^b columns. The entry in row r and column c of the routing table contains a random nodeId that shares the first r digits with the local node's nodeId, and has the $(r + 1)$th digit equal to c. If there is no such nodeId, the entry is left empty. The uniform random distribution of nodeIds ensures that only $log_{2^b} N$ rows have non-empty entries on average. Additionally, the column in row r corresponding to the value of the $(r + 1)$th digit of the local node's nodeId remains empty.

Nodes use a *neighbour selection function* to select between two candidates for the same routing table slot. Given two candidates y and z for slot (r, c) in node x's routing table, x selects z if z's nodeId is numerically closer than y's to the nodeId obtained by replacing the $(r + 1)$th digit of x's nodeId by c. This neighbour selection function promotes stability in routing tables while distributing load. We chose not to use proximity neighbour selection because it increases overhead slightly and low delay routes do not seem important for the applications we study in this paper.

The leaf set connects nodes in a ring. It contains the $l/2$ closest nodeIds clockwise from the local nodeId and the $l/2$ closest nodeIds counter clockwise. The leaf set ensures reliable message delivery. We use $l = 32$ in this paper, which provides high robustness to large scale failures and high churn rates.

At each routing step, the local node normally forwards the message to a node whose nodeId shares a prefix with

the key that is at least one digit longer than the prefix that the key shares with the local node's nodeId. If no such node is known, the message is forwarded to a node whose nodeId is numerically closer to the key and shares a prefix with the key at least as long. The leaf set is used to determine the destination node in the last hop.

Exploiting structure to reduce maintenance overhead Structured overlays can use structure to reduce maintenance overhead in several ways. First, several structured overlays use structure to initialize the routing tables of joining nodes efficiently and to announce their arrival.

Node joining in Pastry exploits the topology structure as follows. A joining node x picks a random nodeId X and asks a bootstrap node a to route a special join message using X as the destination key. This message is routed to the node z with nodeId numerically closest to X. The nodes along the overlay route add routing table rows to the message; node x obtains the rth row of its routing table from the node encountered along the route whose nodeId matches x's in the first $r-1$ digits and its leaf set from z. After initializing its routing table, x sends the rth row of the table to each node in that row. This serves both to announce x's presence and to gossip information about nodes that joined previously. Each node that receives a row considers using the new nodes to replace entries in its routing table.

Additionally, structured overlays can eliminate redundant failure detection probes by using structure to partition failure detection responsibility and to locate nodes that need to be informed when a failure is detected. For example, MS Pastry uses this technique to reduce the number of liveness probes in the leaf set by a factor of 32. Each node sends a single *I'm alive* message every t_l seconds to its left neighbour in the id space. If a node does not receive a message from its right neighbour, it probes the neighbour and marks it faulty if it does not reply. When it marks the neighbour faulty, it discovers the new member of its leaf set by querying the right neighbour of the failed node and informs all the members of the new leaf set about the failed node. If several consecutive nodes in the ring fail, the left neighbour of the leftmost node will detect the failure and repair provided the number of consecutive nodes that failed is less than $l/2-1$. We use $t_l = 30$ seconds in this paper, which is equal to the period between *I'm alive* messages in the unstructured overlays. This technique is readily applicable to systems that organize nodes into a logical ring, for example [32, 29, 28], but harder to apply to other systems, for example [25, 35].

The technique can be extended to eliminate fault detection probes sent to routing table entries. This can be done in routing tables that constrain each node x to point to nodes whose identifiers are the closest to specific points in the identifier space derived from x's nodeId, for example, the original Chord [32] finger table and Pastry's constrained routing table [7]. For example, Pastry's constrained routing table enables a node that detects the failure of its right neighbour to locate all nodes with routing table entries pointing to the failed node with an expected cost of O(log N) messages. We chose not to use the constrained routing table because it eliminates the flexibility necessary to cope with heterogeneous peers as described in the next section.

MS Pastry uses a different strategy to detect failures in the routing table. Since the routing table is not symmetrical, a node explicitly probes every member every t_r seconds to detect failures. The routing table probing period t_r is set dynamically by each node based on the node failure rate in the overlay observed by the node [6]. We configured MS Pastry to achieve a 1% loss rate, i.e., a message routed between a pair of nodes has a probability of 99% of reaching the destination even in the absence of retransmissions.

Pastry also has a *periodic routing table maintenance* protocol to repair failed entries. Each node x asks a node in each row of the routing table for the corresponding row in its routing table. x chooses between the new entries in received rows and the entries in its routing table using the neighbour selection function defined above. This is repeated periodically, for example, every 20 minutes in the current implementation. Additionally, Pastry has a *passive routing table repair* protocol: when a routing table slot is found empty during routing, the next hop node is asked to return any entry it may have for that slot.

These techniques used to reduce overhead in MS Pastry are described in detail in [6] and are applicable to other structured overlays.

2.3 Experimental comparison

We compare the maintenance overhead of the different overlays using a packet-level discrete-event simulator. We simulated a transit-stub network topology [34] with 5050 routers. There are 10 transit domains at the top level with an average of 5 routers in each. Each transit router has an average of 10 stub domains attached, and each stub has an average of 10 routers. Routing is performed using the routing policy weights of the topology generator [34]. The simulator models the propagation delay on the physical links. The average delay of router-router links was 40.7ms. In the experiments, each end system node was attached to a randomly selected stub router with a link delay of 1ms.

The simulation is driven using a real-world trace of node arrivals and failures from a measurement study of Gnutella [30]. The study monitored 17,000 unique nodes in the Gnutella overlay over a period of 60 hours. It probed each node every seven minutes to check if it was still part of the overlay. The average session time over

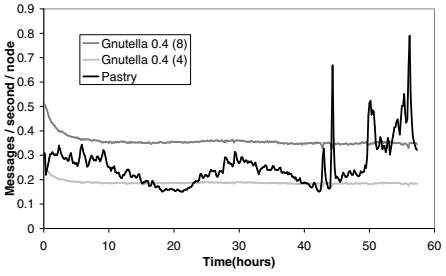


Figure 1: Maintenance overhead in messages per second per node over time for the *Gnutella 0.4* and Pastry overlays.

the trace was approximately 2.3 hours and the number of active nodes in the overlay varied between 1,300 and 2,700. The failure rate and arrival rates are similar but there are large daily variations (more than a factor of 3). There was no application-level traffic during this experiment to isolate the overlay maintenance overhead.

We opted for a simulation study because scalability is an important attribute of these overlays and the testbeds we have available cannot cope with the overlay sizes that we simulate in this and later sections. The code that runs in the simulator is complete and realistic; it can run in a real deployment by simply relinking with a different communication library. The simulator also appears to be accurate as shown by the validation study presented in [6], which compares the simulator output with values measured in a real deployment.

We compare the maintenance overhead of Gnutella 0.4 and Pastry. We used two configurations of Gnutella 0.4: *Gnutella 0.4 (4)* bounds the number of neighbours to be at least 4 and no more than 12, *Gnutella 0.4 (8)* bounds the number of neighbours to be at least 8 and no more than 32. In the experiments, we observed that *Gnutella 0.4 (4)* has on average 5.8 neighbours and *Gnutella 0.4 (8)* has on average 11.0 neighbours.

These parameters were chosen because *Gnutella 0.4 (4)* has maintenance overhead lower than Pastry whereas *Gnutella 0.4 (8)* has higher overhead. It is important to note that both configurations have lower resilience to churn than Pastry. Each Pastry node has 32 neighbours in the leaf set alone and it detects and repairs failures of leaf set neighbours as fast as the Gnutella overlays detect and repair their neighbour failures. A node only gets partitioned from the overlay if 32 nodes fail before being replaced in Pastry whereas it only takes 6 nodes to fail in *Gnutella 0.4 (4)* and 11 in *Gnutella 0.4 (8)*.

Figure 1 shows the maintenance overhead measured as the average number of messages per second per node. The x-axis represents simulation time.

Most of the overhead is due to fault detection messages in the three overlays. In the Gnutella overlay, nodes send *I'm alive* messages to each of their neighbours every 30 seconds. The average number of links per node over the trace is 5.8 in *Gnutella 0.4 (4)* and 11.0 in *Gnutella 0.4 (8)*. Therefore, the expected overhead due to fault detection is 0.19 and 0.37 messages per second per node in *Gnutella 0.4 (4)* and *Gnutella 0.4 (8)*, respectively. Pastry's maintenance overhead is between the overhead of *Gnutella 0.4 (4)* and *Gnutella 0.4 (8)* most of the time.

Pastry is able to achieve low maintenance overhead because it exploits structure. The overhead for fault detection of leaf set members is only 0.03 messages per second per node even though there are 32 nodes in each node's leaf set. Additionally, Pastry tunes the routing table probing period to achieve 1% loss rate (using the techniques described in [6]). This ensures that it uses the minimum probe rate that achieves the desired reliability. Pastry's maintenance overhead varies with the failure rate observed during the trace because the self-tuning technique increases the probe rate when the node failure rate increases. The spikes in maintenance overhead at approximately 44 hours and after 50 hours are due to spikes in the node failure rate in the trace. These spikes in failure rate are probably caused by temporary loss of network connectivity between the site issuing the pings and a large fraction of its targets during the collection of the trace.

It is possible to lower the overhead of Gnutella by reducing the rate of *I'm alive* messages or the number of neighbours but doing this decreases resilience to churn and degrades search efficiency. It might also be possible to use techniques similar to Pastry's to reduce maintenance overhead in Gnutella overlays without decreasing resilience but this would require introducing a structure similar to Pastry's. However, this is not the point.

The important point is that the maintenance overhead is negligible in all three systems and that structured overlays provide additional functionality that has proven useful in a number of applications. For example, the average number of messages per second per node over the trace is only 0.26 in Pastry. Furthermore, the vast majority of these messages are smaller than 100 bytes on the wire. Therefore, the overhead is less than 26 bytes per second, which is negligible even for users with slow dialup connections. For comparison, the latest Gnutella specification [2] recommends a probing period that results in an estimated 131 bytes per second *per neighbour*.

The maintenance overhead is constant in the unstructured overlays but grows with N in the structured overlay. However, it grows very slowly. The fault detection traffic, which accounts for most of the maintenance overhead, is constant for leaf set members and it is proportional to $log_2(N)$ for routing table entries. For example, increasing N to one billion nodes with a similar pattern of node arrivals and departures would increase maintenance traffic in the structured overlay to less than 0.69

messages per second per node (or less than 69 bytes per second per node), which is still negligible.

3 Exploiting heterogeneity

Nodes in deployed peer-to-peer overlays are heterogeneous [30]; they have different bandwidth, storage, and processing capacities. An overlay that ignores the different node capacities must bound the load on any node to be below the load that the least capable nodes are able to sustain; otherwise, it risks congestion collapse. It is important to exploit heterogeneity to improve scalability.

Can unstructured overlays exploit heterogeneity more effectively than structured overlays?

Structured overlays have constraints on the graph topology that reduce flexibility to adapt the topology to exploit heterogeneity. However, some structured overlays have significant flexibility in the choice of some overlay neighbours, which is important to implement proximity neighbour selection [35, 29, 16, 28]. These structured overlays can exploit heterogeneity by modifying the proximity neighbour selection algorithm to choose nodes with high capacity as overlay neighbours. We show that this is as effective as recent proposals to adapt unstructured overlay topologies [10].

This section describes the implementation of several structured and unstructured overlay maintenance protocols that exploit heterogeneity and compares their performance.

3.1 Unstructured overlays

We implemented two unstructured overlay maintenance algorithms that exploit heterogeneity: a version of *Gnutella 0.6* [2] and a version of Gia [10].

Gnutella 0.6 extends the Gnutella 0.4 protocol by adding the concept of super-peers [3]. Nodes that are capable of contributing enough resources to the overlay are classified as super-peers and organized into a random graph using the optimized version of the Gnutella 0.4 protocol (which was described in the previous section). Ordinary nodes are not part of the random graph. Instead, each ordinary node attaches to a small number of randomly selected super-peers and proxies its data discovery queries through them. Ordinary nodes select super-peers to attach to using a random walk with a modified neighbour discovery message and they exchange *I'm alive* messages with the selected super-peers to detect failures. This topology places most of the search and overlay maintenance load on super-peers.

Gia [10] provides a more fine-grained adaptation to heterogeneity. Each node selects a numerical *capacity* value that abstracts the amount of resources that it is willing to contribute to the overlay. Gia adapts the overlay topology such that nodes with higher capacity have higher degree. Since high-degree nodes receive a larger fraction of the traffic, this ensures that they have the capacity to handle this traffic. Gia's fine-grained approach to exploit heterogeneity can perform better than simply using super-peers [10].

We implemented Gia exactly as described in [10]. Node discovery is implemented using a random walk (as described for Gnutella 0.4) but the nodes use Gia's *pick_neighbor_to_drop* function [10] to decide whether to send back a neighbour invitation message. Topology adaptation is driven by Gia's *satisfaction_level* function, which increases with the sum of the ratio between the capacity and degree of each neighbour. This function is evaluated periodically and nodes with a low satisfaction level attempt to find a new neighbour to increase the level. The adaptation interval is computed as in Gia (with the parameters $K = 256$ and $T = 10$ seconds).

3.2 Structured overlays

We implemented two structured overlay maintenance protocols based on Pastry that exploit heterogeneity: *SuperPastry* uses super-peers like Gnutella 0.6 and *HeteroPastry* uses topology adaptation like Gia.

It is simple to exploit the super-peers concept in a structured overlay. The super-peers are organized into a structured overlay using the Pastry algorithm described in the previous section. Ordinary peers do not join this overlay. Instead they attach to a small number of super-peers as in Gnutella 0.6. Ordinary peers select super-peers to attach to by routing to random destination keys through a bootstrap super-peer. They exchange *I'm alive* messages with the selected super-peers to detect failures as in Gnutella 0.6.

The implementation of capacity-aware topology adaptation in structured overlays is less obvious. We propose a simple solution based on existing proximity neighbour selection algorithms [29, 35, 16]. These algorithms select the closest neighbours in the underlying network subject to the structural constraints on the topology. They can be modified to provide capacity-aware topology adaptation by using a proximity metric that reflects node capacity.

HeteroPastry uses the Pastry algorithm described in the previous section except that it achieves capacity-aware topology adaptation by modifying the neighbour selection function to take node capacity into account. Given two candidates y and z for slot (r, c) in node x's routing table, x selects z if it has capacity greater than y or if z and y have the same capacity and z's nodeId is numerically closer than y's to the nodeId obtained by replacing the $(r+1)$th digit of x's nodeId by c. We assume that node capacities are quantized into a few discrete values for the randomization based on nodeIds to be effec-

tive at distributing load. It is possible to design neighbour selection functions that combine several capacity metrics and even network proximity.

In addition to specifying capacity, nodes can specify an upper bound on their indegree, i.e., the number of nodes with routing table entries pointing to them. This bound is likely to be a function of their capacity. We modified Pastry to ensure that the number of routing table entries pointing to a node does not exceed the specified bound. Each node x keeps track of nodes with routing table entries that point to x (backpointers) and sends *backoff* messages when the number of backpointers exceeds the indegree bound. It is necessary to keep track of backpointers because neighbour links in Pastry routing tables are not symmetric. Neighbour links in the leaf set are symmetric and their number is fixed at 32 in this paper. They are not counted as part of the indegree of x unless they also have a routing table entry pointing to x.

Nodes keep track of backpointers by passively monitoring messages received from other nodes. They add a node to the backpointer set when they receive a message from the node and, every D seconds, they remove nodes from which they did not receive messages for more than $2D$ seconds. D is set to the routing table probing period because nodes send probes to their routing table entries every routing table period.

If the number of backpointers exceeds the bound after adding a new node, the local node x selects one of the backpointers for removal and sends that node a backoff message. For each backpointer y with x in slot (r, c) of its routing table, the numerical distance between x's nodeId and the nodeId obtained by replacing the $(r+1)$th digit of y's nodeId by c is computed. x selects the node with the maximal distance for eviction. This policy is dual of the neighbour selection function (except that it is oblivious to capacity) to provide stability.

Nodes that receive a backoff message remove the sender from their routing tables and insert the sender in a backoff cache. We modified the neighbour selection function to ensure that it never selects nodes in the backoff cache. The current implementation removes entries from the backoff cache after four routing table probing periods.

Our solution is not applicable to some structured overlays that provide no flexibility at all in the selection of neighbours, for example, the original Chord [32] and CAN [25]. It is possible to use virtual nodes [32] to adapt these structured overlays to different node capacities. Each physical node can simulate a number of virtual overlay nodes proportional to its capacity. The problem is that node capacities can vary by several order of magnitude. Therefore, the number of virtual nodes must be much larger than the number of physical nodes, which results in a large increase in maintenance traffic that can render this solution impractical.

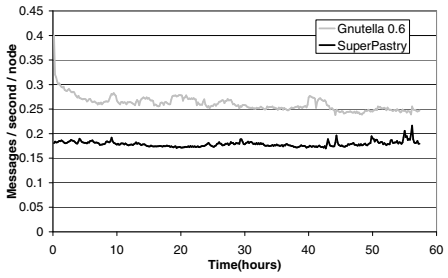


Figure 2: Maintenance overhead in messages per second per node over time for the two overlays using super-peers.

3.3 Experimental comparison

We compared the maintenance overhead of the different overlay maintenance algorithms that exploit heterogeneity to achieve scalability. We used the experimental setup in Section 2.3, which does not include any query traffic, to isolate the maintenance overheads.

Gnutella 0.6 and SuperPastry were configured with similar parameters to allow a fair comparison. Each ordinary node selected 3 super-peers as proxies and each super-peer acted as a proxy for up to 30 ordinary nodes. Each super-peer in Gnutella 0.6 had at least 10 super-peer neighbours and at most 32. The indegree bound of super-peers in SuperPastry was also 32. The simulator provided each joining node with a randomly selected super-peer to bootstrap the joining process and joining nodes were marked super-peers with a probability of 0.2. Figure 2 shows the maintenance overhead measured as the number of messages sent per second per node.

The maintenance overhead is dominated by the cost of failure detection as before. In Gnutella 0.6, a node has 7.5 neighbours on average, which results in 0.25 *I'm alive* messages per second per node on average. This accounts for most of the control traffic has shown in Figure 2. Both systems incur the same communication overhead between ordinary peers and super-peers. SuperPastry achieves lower overhead than Gnutella 0.6 because it exploits structure to reduce failure detection overhead. The overhead is negligible in both systems.

We also ran experiments to compare the maintenance overhead of Gia and HeteroPastry. Gia was configured using the parameters in [10]. The lower bound on the number of neighbours in Gia is 3 and the upper bound is $max(3, min(128, \frac{C}{4}))$ [10], where C is the capacity of the node. We use the same bounds on the indegree of nodes in HeteroPastry. The capacity of a node (in both overlays) is selected when it joins according to the probabilities in Table 1, which were taken from [10].

Figure 3 plots the maintenance overhead in messages per second per node against time for Gia and HeteroPas-

Capacity	Probability
1	0.2
10	0.45
100	0.3
1000	0.049
10000	0.001

Table 1: Node capacity distribution

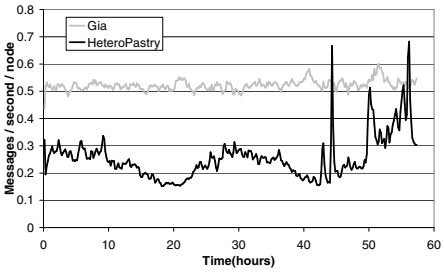


Figure 3: Maintenance overhead in messages per second per node over time for Gia and HeteroPastry.

try. Failure detection messages account for most of the overhead as in previous experiments. Nodes in Gia have 15.6 neighbours on average, which results in 0.52 *I'm alive* messages per second per node. The overhead of HeteroPastry is almost identical to the overhead incurred by the version of Pastry that does not exploit heterogeneity and does not bound indegrees (which is shown in Figure 1).

Figure 3 shows that the overhead of topology adaptation in both Gia and HeteroPastry is negligible. The next set of results show that topology adaptation in HeteroPastry is also effective.

We examined the routing tables of live HeteroPastry nodes five hours into the trace and calculated the average capacity of the nodes in routing table entries at each routing table level across the 2627 live nodes. Figure 4 shows the results.

Topology adaptation fills routing tables with high capacity nodes. The average capacity of nodes in levels up

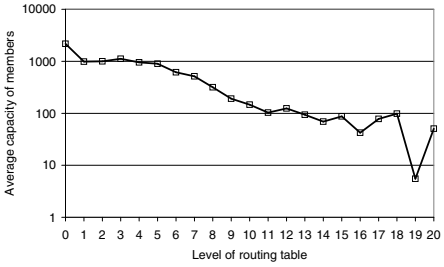


Figure 4: Average capacity of nodes in routing table entries at each level in HeteroPastry.

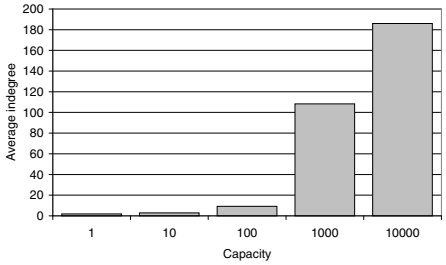


Figure 5: Average indegree of nodes with each capacity value.

to 5 is above 897. The capacity decreases when the level increases because of stronger structural constraints. A node in level l of the routing table must match the nodeId of the local node in the first l digits. The size of the set of nodes that can be selected to fill slots at level $l+1$ is half the size of the set of nodes that can fill slots at level l. Therefore, the probability that these sets include high capacity nodes decreases as the level increases. Since most nodes have less than 12 ($log_2(2627)$) levels in their routing tables, there is some noise for levels above 12.

We also measured the average indegree of nodes with each capacity value at the same point in time. The results are in Figure 5. The average indegree of the two nodes with capacity 10000 is above the indegree bound of 128. This happens because nodes are very likely to select nodes with capacity 10000 for the top levels of their routing tables and these pointers are only removed after the node receives a backoff message. The results show that topology adaptation in HeteroPastry is effective at distributing the indegree according to capacity.

4 Data queries

Complex queries are important in mass-market data sharing applications [10]. Since users do not know the exact names of the files they want to retrieve, the exact-match queries offered by structured overlays are not directly useful in these applications. Users discover data with keyword searches, which are readily supported by unstructured overlays that visit a subset of random nodes in the overlay and execute the search query locally at each visited node.

Can unstructured overlays support complex queries more efficiently than structured overlays?

Several research prototypes support keyword searches using the exact-match queries of structured overlays [27, 33, 14, 18] to implement inverted indices. The basic idea is to use the structured overlay to map keywords to overlay nodes. The node responsible for a keyword stores an index with the location of all documents that contain the keyword. When a file is added to the system, the nodes

responsible for the keywords in the file are contacted to update the appropriate indices. A query for documents containing a set of keywords contacts the nodes responsible for those keywords and intersects their indices.

Unfortunately, this approach has several problems. Maintaining the indices in the presence of churn is expensive and popular keywords may be mapped to low capacity nodes that cannot cope with the load [10]. Additionally, the queries can be expensive because they require computing the intersection of large indices. The analysis in [20] shows that this approach performs worse than flooding queries to 60,000 nodes in a random graph. Therefore, this approach performs significantly worse than recent unstructured overlays like Gia [10]. Additionally, unstructured overlays can support even more sophisticated queries that are not supported by the inverted indices approach, for example, regular expressions and range queries on multiple attributes.

This section explores a different approach to supporting complex queries in structured overlays. We developed a hybrid system that uses the topology from structured overlays with the data placement and data discovery strategies of unstructured overlays. We introduce new techniques to perform floods or random walks over structured topologies that provide support for arbitrarily complex queries. These techniques take advantage of structural constraints on the topology to ensure that nodes are visited only once during a query, to control the number of nodes that are visited accurately, and to increase the average capacity of nodes visited during a query to exploit heterogeneity more effectively.

The results in the previous sections show that it is possible to maintain a structured overlay that exploits heterogeneity with low maintenance overhead. Additionally, the hybrid system does not constrain data placement; nodes do not have to incur the overhead of updating distributed indices for each keyword in their files. This section compares the performance of random walks and floods on the overlays that were described in the previous section.

4.1 Unstructured overlays

We used random walks to discover data because they have been shown to induce lower overhead than the constrained floods [23] used by current versions of Gnutella. These random walks are biased to prefer nodes with higher degree in Gia and are unbiased in the other unstructured overlays. The original Gia [10] biased the random walks to prefer nodes with higher capacity but our experimental results indicate that preferring nodes with higher degree yields both higher success rate and lower delay. We present results for this optimized version of Gia.

We observed that random walks in Gia were likely to visit the same node more than once, which resulted in worse search performance. We added a list to each query with all the nodes already visited by the query to prevent this. Nodes do not forward a query to a node that is in this list.

All unstructured overlays use *one hop replication*, which has been shown to improve search performance in unstructured overlays [10]. A node replicates an index of its content at each of its neighbours. In Gnutella 0.6, these indices are only replicated at super peers.

4.2 Structured Overlays

The hybrid system exploits structure to implement random walks and constrained floods more efficiently.

Flooding in random graphs is inefficient because each node is likely to be visited more than once. In a graph with an average degree of k, a flood that visits all nodes will send on average $(k-1) \times N$ messages (where N is the size of the overlay). Additionally, it is difficult to control the number of nodes visited during a constrained flood. Floods are constrained using a time-to-live field in the query message that is decremented every time the query is forwarded. The query is not forwarded when the time-to-live field drops to zero. This provides very coarse control over the number of nodes visited.

The hybrid system can do better by replacing flooding with the broadcast mechanisms that have been proposed for structured overlays [26, 9, 11]. We use Pastry's broadcast mechanism [9] to flood queries to overlay nodes. A node y broadcasts a query by sending the query to all the nodes x in its routing table. Each query is tagged with the routing table row r of node x. When a node receives a query tagged with r, it forwards the query to all nodes in its routing table in rows greater than r if any.

A node may have a missing entry in a slot in its routing table, for example, because it pointed to a node that failed. The broadcast overcomes this problem by using Pastry to route the query to a node with the appropriate nodeId to fill the slot (if there is any) [9]. Almost all nodes receive the query only once but the technique to deal with empty routing table slots may result in a small number of duplicates.

We place an upper bound on the row number of entries to which the query is forwarded to constrain the flood. This bounds the number of nodes visited to a power of two. It is simple to extend this mechanism to provide arbitrarily fine grained control over the number of nodes visited.

This mechanism can easily be modified to perform random walks rather than floods by performing a breadth first traversal of the tree used for flooding. This can be done by adding a set of nodes to visit in the query message. A random walk query message includes the tag r,

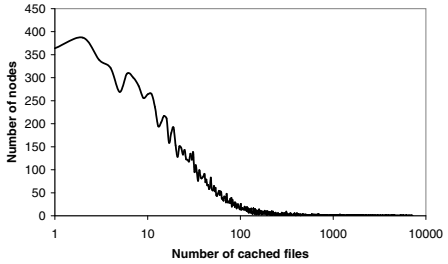


Figure 6: Distribution of the number of files per node for the eDonkey file trace [12].

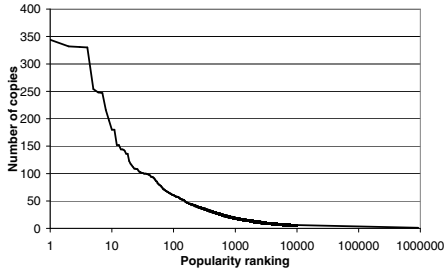


Figure 7: Number of files versus file rank for the eDonkey file trace [12].

an array q with queues of nodes indexed by routing table row, and a bound d on the maximum row number to traverse. When the query is received at node x, it appends the nodes in each routing table row r' to queue $q[r']$ provided that $r < r' \leq d$. Then, if queue $q[r]$ is not empty, x removes the next node from the queue and forwards the query to this node. If $q[r]$ is empty, the query is forwarded to the first node in queue $q[r+1]$ and r is incremented. If all queues are empty, the random walk is complete.

The results in the previous section show that the average capacity of the nodes in routing table entries in HeteroPastry decreases as the row number increases. Therefore, the mechanism that we use to bound the floods and random walks biases them to visit nodes with higher capacity in HeteroPastry.

We also implement *one hop replication* in the hybrid system. Each node replicates an index of its local content on the nodes in its routing table. Therefore, it is expected to replicate its index in $log_2(N)$ other nodes.

4.3 Experimental comparison

We compared the performance of random walks on structured and unstructured overlays. We used the basic experimental setup described in the previous sections but we simulated queries and node file stores.

We used a real-world trace of files stored by eDonkey [12] peers to model the sets of files stored by simulated nodes. There are 37,000 peers in the trace and, for each peer, there is a record with the identifiers of the files stored by the peer. Figure 6 shows the distribution of the number of files stored by each peer. It excludes the 25,172 peers that have no files. We model the set of files stored by each node as follows: when a node joins, the simulator chooses a random unused record from the trace and assigns the files in the record to the node.

There are approximately 923,000 unique files. File copies exhibit a heavy-tailed zipf-like distribution as shown in Figure 7. Full details about the trace can be found in [12].

The eDonkey trace does not include queries but the number of copies of a file is strongly correlated with the number of queries that it satisfies. Therefore, our query distribution matches the distribution of the number of copies of files.

Each node generates 0.01 query messages per second using a Poisson process and each query searches for a file in the trace. The simulator maintains the distribution of the number of copies of files stored by nodes that are currently in the overlay. The target file for each query is chosen from this distribution (which is a sample of the distribution in Figure 7). This ensures that at least one copy of the target file is stored in the overlay when the query is initiated.

In all the experiments, we bound random walks to visit at most 128 nodes. When a node x receives a query, it checks if the target file is stored locally or if it is stored by nodes whose indices are replicated locally. In the first case, the query is satisfied and x does not forward the query further. In the second case, x contacts a random node y which it believes has a copy of the file. If y has the file, the query is satisfied and y sends an acknowledgment back to x. If x receives the acknowledgment before a timeout, it stops forwarding the query. Otherwise, x contacts another random node that it believes has the file or it forwards the query if there are no more such nodes.

We measured the fraction of queries that are satisfied and the delay from the moment a query is initiated until it is satisfied. We also measured the load by counting the number of messages sent per second per node.

4.3.1 Gnutella trace

We compared the performance of data discovery on the overlays that exploit heterogeneity. Figure 8 shows the query success rate, Figure 9 shows the delay for successful queries, and Figure 10 shows the overhead in messages per second per node. The results show that fine-grained topology adaptation performs better than using super-peers. HeteroPastry achieves significantly higher

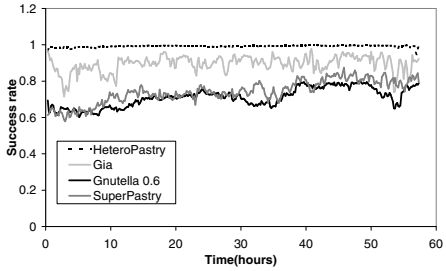


Figure 8: Query success rate.

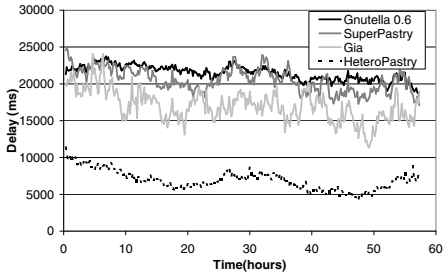


Figure 9: Query delay for successful queries.

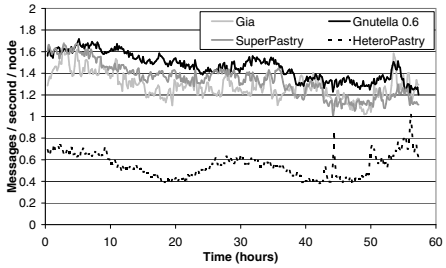


Figure 10: Messages per second per node.

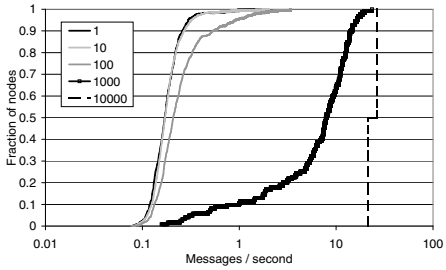


Figure 11: Cumulative distribution of messages per second per node for each capacity value in HeteroPastry.

success rate, and lower delay and overhead than SuperPastry and Pastry. We also ran experiments with overlays that do not exploit heterogeneity and found that they perform significantly worse.

SuperPastry and Gnutella 0.6 achieve very similar performance by all metrics. But HeteroPastry achieves significantly better performance than all the others. It achieves the highest success rate, the lowest delay, and the lowest overhead. This demonstrates that HeteroPastry can exploit heterogeneity effectively to improve scalability; the high success rate indicates that the bound on the length of random walks can be small and the low delay shows that they are likely to terminate early, which results in low overhead. The other systems would require longer random walks to achieve the success rate of HeteroPastry, which would increase their overhead.

All the overlay maintenance algorithms benefit from suppression of failure detection traffic by query traffic. For example, Gia's overhead without queries is approximately twice the overhead of Gnutella 0.6. The overheads of the two are comparable with queries because of the suppression of failure detection traffic and shorter random walks.

So far we have considered the overhead averaged over all live nodes in each 10 minute window in the trace. Since both Gia and HeteroPastry adapt the topology to distribute load according to node capacity, we looked at the distribution of the number of messages per second per node in the ten minutes preceding the 5 hour mark in the trace. The total number of messages received in this 10 minute window was 2.4 times higher for Gia than HeteroPastry. Figures 11 and 12 show the cumulative distribution of the number of messages per second per node for each capacity value in HeteroPastry and Gia.

The maximum message rate observed was only 42.63 for Gia and 26.48 for HeteroPastry. Both systems do a good job of distributing message load according to capacity; nodes with higher capacity receive more messages. The message rate for nodes with capacity 1 is low; the median is only 0.17 and the 95th percentile is only 0.30 in HeteroPastry, and the median is 0.11 and the 95th percentile is 0.13 in Gia. For the nodes with capacity 10 in HeteroPastry, the median is also 0.17 and the 95th percentile is 0.32, and the median is 0.11 and the 95th percentile is 0.14 in Gia. Since the indegree of 1-

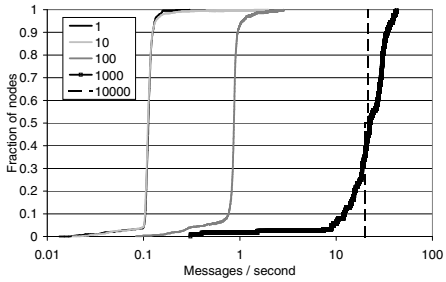


Figure 12: Cumulative distribution of messages per second per node for each capacity value in Gia.

	Capacity	1	10	100	1000	10000
Gia	Mean	3	3	23.56	126.02	128
	Median	3	3	24	128	128
	95th	3	3	25	128	128
Hetero-Pastry	Mean	2.15	2.38	14.50	104.66	128
	Median	2	3	15	125	128
	95th	3	3	24	128	128

Table 2: Distribution of replicas of node indices for different capacity values in Gia and HeteroPastry.

and 10-capacity nodes is bounded to the same value, this is not surprising. In both Gia and HeteroPastry, the 100-capacity nodes incur a higher overhead than the 1- and 10-capacity nodes but a lower overhead than the 1000-capacity nodes.

The figures also show that the load on any node is sufficiently low (with a query rate of 0.01 queries per second per node) that flow control is not necessary. Gia's flow control mechanism [10] can be applied to HeteroPastry to enable scaling to higher query rates.

We also studied the distribution of replicas of node indices, which is another indicator of the effectiveness of both systems in adapting the topology to diffferent node capacities. Table 2 summarises the distribution of replicas of indices for each capacity value in both systems. The total numbers of index replicas is 27,707 in HeteroPastry and 38,153 in Gia. Both systems do a good job at distributing index replicas (and indegree) according to node capacity. Gia replicates more because it is more effective at pushing replicas to nodes with capacity 100 and 1000.

HeteroPastry maintains significantly less index replicas than Gia but it performs better because its random walks visit nodes with more index replicas and more diverse index replicas than those visited by random walks in Gia. In Gia, nodes that are close in the overlay topology tend to share the same high capacity neighbours. This reduces the number of unique files known by a node and its neighbours and it forces biased random walks to visit low capacity nodes before they can find new high capacity nodes to visit. Since the number of index replicas stored by a node is proportional to its capacity, this results in poor performance. The topology adaptation and random walk mechanisms in HeteroPastry exploit structure to prevent this problem; the constraints on the node identifiers of neighbours and nodes visited during a random walk ensure that the initial set of nodes visited has high capacity and knows about more unique files. This results in HeteroPastry visiting significantly less nodes with capacity 100 during random walks than Gia (as shown in Figure 11).

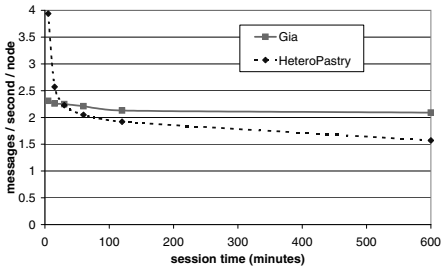


Figure 13: Messages per second per node for Gia and HeteroPastry versus session time.

4.3.2 Poisson traces

The experiments described so far use a trace of node arrivals and departures collected in a real Gnutella deployment. The next set of experiments compare the performance of Gia and HeteroPastry using artificial traces with more nodes and different rates of churn. These traces have Poisson node arrivals and an exponential distribution of node session times with the same rate. We generated traces with session times of 5, 15, 30, 60, 120 and 600 minutes and in all cases the average number of nodes was 10,000. We used the same data and query distribution as in the previous experiments. It is important to note that a session time of 5 minutes is short; indeed, it is 28 times shorter than the average session time of 2.3 hours observed in the Gnutella trace.

Figure 13 shows the total number of messages per second per node for the different session times. Both Gia and HeteroPastry have low overhead across all session times.

Gia's overhead is almost constant across all session times. Short session times increase Gia's overhead because of increased retransmissions and traffic to fill neighbour tables. However, this is offset by a decrease in fault detection traffic due to a decrease in the average number of neighbours; there are 15.1 neighbours when the session time is 600 and 10.7 when it is 5.

HeteroPastry has a lower message overhead than Gia for session times of 30 minutes or greater. This overhead decreases between 60 and 600 minutes because HeteroPastry adapts the routing table probing rate to match the failure rate. HeteroPastry incurs a higher message overhead than Gia for extremely high churn rates mostly due to the overhead of maintaining the leaf set. This overhead could be reduced without impacting query success rate and delay by using a smaller leaf set or disabling the mechanisms to ensure strong leaf set consistency [6], which are not important in this application.

Figure 14 shows the lookup success rate for the different session times. As in previous experiments, HeteroPastry achieves a success rate higher than Gia across all session times.

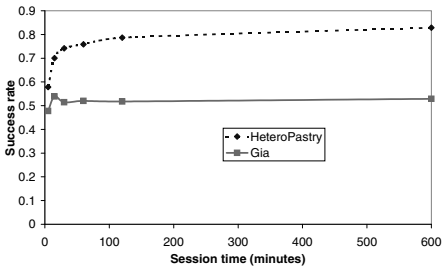


Figure 14: Query success rate for Gia and HeteroPastry versus session time.

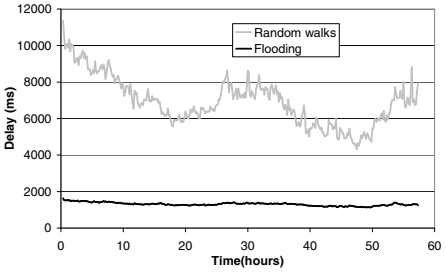


Figure 15: Query delay when using constrained flooding and random walks in HeteroPastry.

The success rates with 10,000 nodes are lower than those observed before because there are more nodes and random walk length is still bound to 128. There are at most 2,700 active nodes at any time in the Gnutella trace. This also results in higher message overhead with 10,000 nodes even with a session time of 600 minutes.

The delay incurred for successful lookups is similar in both HeteroPastry and Gia. HeteroPastry achieves a lower average delay per lookup because it has a higher success rate and failed lookups take longer to complete on average than successful lookups. Therefore, HeteroPastry achieves a delay at least 12% lower than Gia with 5 minute session times and at least 43% lower with 600 minutes session time.

4.3.3 Constrained floods

We also compared the performance of constrained flooding and random walks in HeteroPastry. We configured constrained floods to visit at most 128 nodes as with the random walks. Both algorithms visit exactly the same 128 nodes when the query fails so they have the same success rate.

Figure 15 shows the delay for successful queries using both constrained floods and random walks. It shows that constrained flooding can locate content faster than random walks. This is not surprising because constrained flooding visits nodes in parallel; all 128 nodes are visited after only 7 hops. It takes 128 hops to visit all

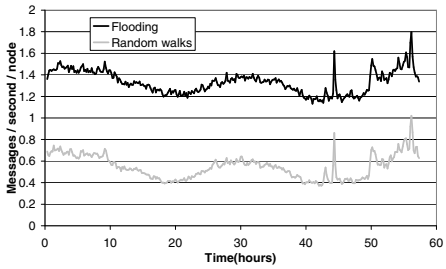


Figure 16: Messages per second per node when using constrained floods and random walks in HeteroPastry.

the nodes with the random walk. Additionally, random walks use acknowledgments and retransmissions to recover when the query is forwarded to a node that fails. This introduces delays that increase when the failure rate in the trace increases (as shown in Figure 15). The delay of constrained floods remains constant because we do not use acknowledgments and retransmissions and instead rely on redundancy to cope with node failures. We observed the same success rate for both flooding and random walks, which demonstrates the effectiveness of using redundancy to cope with node failure during constrained floods.

Figure 16 shows the number of messages per second per node when using constrained floods and random walks in HeteroPastry. It demonstrates the advantage of random walks over flooding; random walks result in lower overhead because they stop when they find a copy of the file and visit less nodes than constrained floods on average. It is interesting to note that the overhead with constrained floods is comparable to the overhead in the unstructured overlays. Additionally, some peer-to-peer applications discover multiple nodes with matching content, for example, to enable more efficient downloads with some form of striping. The benefit of random walks over constrained floods decreases in this case. Constrained floods are likely to be the best strategy for many applications.

5 Conclusion

It is commonly believed that unstructured overlays cope with churn better, exploit heterogeneity more effectively, and support complex queries more efficiently than structured overlays. This paper shows that coping with churn, exploiting heterogeneity and supporting complex queries are not fundamental problems for structured overlays.

We describe how to exploit structure to achieve high resilience to churn with maintenance overhead as low as unstructured overlays and how to modify proximity neighbour selection to exploit heterogeneity effectively to improve scalability. Additionally, we present a hybrid

system that uses the search and data placement strategies of unstructured overlays on a structured overlay topology. Simulation results using a real-world trace show that the hybrid system can support complex queries with lower message overhead while providing higher query success rates and lower response times than the state of the art in unstructured overlays.

The additional functionality provided by structured overlays has proven important to achieve scalability and efficiency in a wide range of applications. Structured overlays can emulate the functionality of unstructured overlays with comparable or even better performance. Interestingly, it is not clear that unstructured overlays can efficiently emulate the same functionality as structured overlays.

References

[1] The Gnutella 0.4 protocol specification, 2000. http://dss.clip2.com/GnutellaProtocol04.pdf.

[2] The Gnutella 0.6 protocol specification, 2002. http://www.limewire.org/.

[3] Kazaa, 2002. http://www.kazaa.com/.

[4] BHAGWAN, R., SAVAGE, S., AND VOELKER, G. Understanding availability. In *IPTPS'03* (Feb. 2003).

[5] BLAKE, C., AND RODRIGUES, R. High Availability, Scalable Storage, Dynamic Peer Networks: Pick Two. In *HotOS IX* (May 2003).

[6] CASTRO, M., COSTA, M., AND ROWSTRON, A. Performance and dependability of structured peer-to-peer overlays. In *DSN'04* (June 2004).

[7] CASTRO, M., DRUSCHEL, P., GANESH, A., ROWSTRON, A., AND WALLACH, D. S. Security for structured peer-to-peer overlay networks. In *OSDI'02* (Dec. 2002).

[8] CASTRO, M., DRUSCHEL, P., HU, Y. C., AND ROWSTRON, A. Proximity neighbor selection in tree-based structured peer-to-peer overlays. Tech. Rep. MSR-TR-2003-52, Microsoft Research, Aug. 2003.

[9] CASTRO, M., JONES, M. B., KERMARREC, A.-M., ROWSTRON, A., THEIMER, M., WANG, H., AND WOLMAN, A. An evaluation of scalable application-level multicast built using peer-to-peer overlays. In *Infocom'03* (Apr. 2003).

[10] CHAWATHE, Y., RATNASAMY, S., BRESLAU, L., LANHAM, N., AND SHENKER, S. Making Gnutella-like p2p systems scalable. In *SIGCOMM'03* (Aug. 2003).

[11] EL-ANSARY, S., ALIMA, L. O., BRAND, P., AND HARIDI, S. Efficient broadcast in structured p2p networks. In *IPTPS'03* (Feb. 2003).

[12] FESSANT, F. L., HANDURUKANDE, S., KERMARREC, A.-M., AND MASSOULIE, L. Clustering in peer-to-peer file sharing workloads. In *IPTPS'04* (Feb. 2004).

[13] GANESAN, P., SUN, Q., AND GARCIA-MOLINA, H. Yappers: A peer-to-peer lookup service over arbitrary topology. In *Infocom'03* (Apr. 2003).

[14] GNAWALI, O. A keyword set search system for peer-to-peer networks, 2002. Master Thesis, MIT.

[15] The Gnutella protocol specification, 2000. `http://dss.clip2.com/GnutellaProtocol04.pdf`.

[16] GUMMADI, K. P., GUMMADI, R., GRIBBLE, S. D., RATNASAMY, S., SHENKER, S., AND STOICA, I. The impact of DHT routing geometry on resilience and proximity. In *SIGCOMM'03* (Aug. 2003).

[17] GUMMADI, P. K., DUNN, R. J., SAROIU, S., GRIBBLE, S. D., LEVY, H. M., AND ZAHORJAN, J. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *SOSP'03* (Oct. 2003).

[18] HARREN, M., HELLERSTEIN, J. M., HUEBSCH, R., LOO, B. T., SHENKER, S., AND STOICA, I. Complex queries in DHT-based peer-to-peer networks. In *IPTPS'02* (Mar. 2002).

[19] IYER, S., ROWSTRON, A., AND DRUSCHEL, P. Squirrel: A decentralized peer-to-peer web cache. In *PODC'02* (July 2002).

[20] LI, J., LOO, B. T., HELLERSTEIN, J., KAASHOEK, F., KARGER, D. R., AND MORRIS, R. On the feasibility of peer-to-peer web indexing and search. In *IPTPS'03* (Feb. 2003).

[21] LI, J., STRIBLING, J., GIL, T. M., MORRIS, R., AND KAASHOEK, M. F. Comparing the performance of distributed hash tables under churn. In *IPTPS'04* (Feb. 2004).

[22] LOO, B. T., HELLERSTEIN, J. M., HUEBSCH, R., SHENKER, S., AND STOICA, I. Enhancing P2P file sharing with an Internet-scale query processor. In *VLDB'04* (Sept. 2004).

[23] LV, Q., CAO, P., COHEN, E., LI, K., AND SHENKER, S. Search and replication in unstructured peer-to-peer networks. In *ICS'02* (June 2002).

[24] LV, Q., RATNASAMY, S., AND SHENKER, S. Can heterogeneity make Gnutella scalable? In *IPTPS'02* (Feb. 2002).

[25] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content-addressable network. In *SIGCOMM'01* (Aug. 2001).

[26] RATNASAMY, S., HANDLEY, M., KARP, R., AND SHENKER, S. Application-level multicast using content-addressable networks. In *NGC'01* (Nov. 2001).

[27] REYNOLDS, P., AND VAHDAT, A. Efficient peer-to-peer keyword searching. In *Middleware'03* (Nov. 2003).

[28] RHEA, S., GEELS, D., ROSCOE, T., AND KUBIATOWICZ, J. Handling churn in a DHT. In *USENIX'04* (June 2004).

[29] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware'01* (Nov. 2001).

[30] SAROIU, S., GUMMADI, K., AND GRIBBLE, S. A measurement study of peer-to-peer file sharing systems. In *MMCN'02* (Jan. 2002).

[31] SEN, S., AND WANG, J. Analyzing peer-to-peer traffic across large networks. In *Internet Measurement Workshop* (Nov. 2002).

[32] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for Internet applications. In *SIGCOMM'01* (Aug. 2001).

[33] TANG, C., XU, Z., AND DWARKADAS, S. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *SIGCOMM'03* (Aug. 2003).

[34] ZEGURA, E., CALVERT, K., AND BHATTACHARJEE, S. How to model an internetwork. In *INFOCOM'96* (1996).

[35] ZHAO, B. Y., KUBIATOWICZ, J. D., AND JOSEPH, A. D. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Tech. Rep. UCB-CSD-01-1141, U. C. Berkeley, Apr. 2001.

Bandwidth-efficient management of DHT routing tables

Jinyang Li, Jeremy Stribling, Robert Morris, and M. Frans Kaashoek
MIT Computer Science and Artificial Intelligence Laboratory
{jinyang, strib, rtm, kaashoek}@csail.mit.edu

Abstract

Today an application developer using a distributed hash table (DHT) with n nodes must choose a DHT protocol from the spectrum between $O(1)$ lookup protocols [9, 18] and $O(\log n)$ protocols [20–23, 25, 26]. $O(1)$ protocols achieve low latency lookups on small or low-churn networks because lookups take only a few hops, but incur high maintenance traffic on large or high-churn networks. $O(\log n)$ protocols incur less maintenance traffic on large or high-churn networks but require more lookup hops in small networks. Accordion is a new routing protocol that does not force the developer to make this choice: Accordion adjusts itself to provide the best performance across a range of network sizes and churn rates while staying within a bounded bandwidth budget.

The key challenges in the design of Accordion are the algorithms that choose the routing table's size and content. Each Accordion node learns of new neighbors opportunistically, in a way that causes the density of its neighbors to be inversely proportional to their distance in ID space from the node. This distribution allows Accordion to vary the table size along a continuum while still guaranteeing at most $O(\log n)$ lookup hops. The user-specified bandwidth budget controls the rate at which a node learns about new neighbors. Each node limits its routing table size by evicting neighbors that it judges likely to have failed. High churn (*i.e.*, short node lifetimes) leads to a high eviction rate. The equilibrium between the learning and eviction processes determines the table size.

Simulations show that Accordion maintains an efficient lookup latency versus bandwidth tradeoff over a wider range of operating conditions than existing DHTs.

1 Introduction

Distributed hash tables maintain routing tables used when forwarding lookups. A node's routing table consists of a set of "neighbor" entries, each of which contains the IP address and DHT identifier of some other node. A DHT node must maintain its routing table, both populating it initially and ensuring that the neighbors it refers to are still alive.

Existing DHTs use routing table maintenance algorithms that work best in particular operating environments. Some maintain small routing tables in order to limit the maintenance communication cost [11, 20–23, 25, 26]. Small tables help the DHT scale to many nodes and limit the maintenance required if the node population increases rapidly. The disadvantage of a small routing table is that lookups may take many time-consuming hops, typically $O(\log n)$ in a system with n nodes.

At the other extreme are DHTs that maintain a complete list of nodes in every node's routing table [9, 18]. A large routing table allows single-hop lookups. However, each node must promptly learn about every node that joins or leaves the system, as otherwise lookups are likely to experience frequent timeout delays due to table entries that point to dead nodes. Such timeouts are expensive in terms of increased end-to-end lookup latency [2, 16, 22]. The maintenance traffic needed to avoid timeouts in such a protocol may be large if there are many unstable nodes or the network size is large.

An application developer wishing to use a DHT must choose a protocol between these end points. An $O(1)$ protocol might work well early in the deployment of an application, when the number of nodes is small, but could generate too much maintenance traffic as the application becomes popular or if churn increases. Starting with an $O(\log n)$ protocol would result in unnecessarily low performance on small networks or if churn turns out to be low. While the developer can manually tune a $O(\log n)$ protocol to increase the size of its routing table, such tuning is difficult and workload-dependent [16].

This paper describes a new DHT design, called Accordion, that automatically tunes parameters such as routing table size in order to achieve the best performance. Accordion has a single parameter, a network bandwidth budget, that allows control over the consumption of the resource that is most constrained for typical users. Given the budget, Accordion adapts its behavior across a wide range of network sizes and churn rates to provide low-latency lookups.

This research was conducted as part of the IRIS project (http://project-iris.net/), supported by the National Science Foundation under Cooperative Agreement No. ANI-0225660. Jinyang Li is also supported by a Microsoft Research Fellowship.

The problems that Accordion must solve are how to arrive at the best routing table size in light of the budget and the stability of the node population, how to choose the most effective neighbors to place in the routing table, and how to divide the maintenance budget between acquiring new neighbors and checking the liveness of existing neighbors.

Accordion solves these problems in a unique way. Unlike other protocols, it is not based on a particular data structure such as a hypercube or de Bruijn graph that constrains the number and choice of neighbors. Instead, each node learns of new neighbors as a side-effect of ordinary lookups, but selects them so that the density of its neighbors is inversely proportional to their distance in ID space from the node. This distribution allows Accordion to vary the table size along a continuum while still providing the same worst-case guarantees as traditional $O(\log n)$ protocols. A node's bandwidth budget determines the rate at which a node learns. Each node limits its routing table size by evicting neighbors that it judges likely to have failed: those which have been up for only a short time or have not been heard from for a long time. Therefore, high churn leads to a high eviction rate. The equilibrium between the learning and eviction processes determines the table size.

Performance simulations show that Accordion keeps its maintenance traffic within the budget over a wide range of operating conditions. When bandwidth is plentiful, Accordion provides lookup latencies and maintenance overhead similar to that of OneHop [9]. When bandwidth is scarce, Accordion has lower lookup latency and less maintenance overhead than Chord [5, 25], even when Chord incorporates proximity and has been tuned for the specific workload [16].

The next two sections outline Accordion's design approach and analyze the relationship between maintenance traffic and table size. Section 4 describes the details of the Accordion protocol. Section 5 compares Accordion's performance with that of other DHTs. Section 6 presents related work, and Section 7 concludes.

2 Design Challenges

A DHT's routing table maintenance traffic must fit within the nodes' access link capacities. Most existing designs do not live within this physical constraint. Instead, the amount of maintenance traffic they consume is determined as a side effect of the total number of nodes and the rate of churn. While some protocols (*e.g.*, Bamboo [22] and MSPastry [2]) have mechanisms for limiting maintenance traffic during periods of high churn or congestion, one of the goals of Accordion is to keep this traffic within a budget determined by link capacity or user preference.

Once a DHT node has a maintenance budget, it must decide how to use the budget to minimize lookup latency. This latency depends largely on two factors: the average number of hops per lookup and the average number of timeouts incurred during a lookup. A node can choose to spend its bandwidth budget to aggressively maintain the freshness of a smaller routing table (thus minimizing timeouts), or to look for new nodes to enlarge the table (thus minimizing lookup hops but perhaps risking timeouts). Nodes may also use the budget to issue lookup messages along multiple paths in parallel, to mask the effect of timeouts occurring on any one path. Ultimately, the bandwidth budget's main effect is on the size and contents of the routing table.

Rather than explicitly calculating the best table size based on a given budget and an observed churn rate, Accordion's table size is the result of an equilibrium between two processes: state acquisition and state eviction. The state acquisition process learns about new neighbors; the bigger the budget is, the faster a node can learn, resulting in a bigger table size. The state eviction process deletes routing table entries that are likely to cause lookup timeouts; the higher the churn, the faster a node evicts state. The next section investigates and analyzes budgeted routing table maintenance issues in more depth.

3 Table Maintenance Analysis

In order to design a routing table maintenance process that makes the most effective use of the bandwidth budget, we have to address three technical questions:

1. How do nodes choose neighbors for inclusion in the routing table in order to guarantee at most $O(\log n)$ lookups across a wide range of table sizes?
2. How do nodes choose between active exploration and opportunistic learning (perhaps using parallel lookups) to learn about new neighbors in the most efficient way?
3. How do nodes evict neighbors from the routing table with the most efficient combination of active probing and uptime prediction?

3.1 Routing State Distribution

Each node in a DHT has a unique identifier, typically 128 or 160 random bits generated by a secure hash function. Structured DHT protocols use these identifiers to assign responsibility for portions of the identifier space. A node keeps a routing table that points to other nodes in the network, and forwards a query to a neighbor based on the neighbor's identifier and the lookup key. In this manner, the query gets "closer" to the node responsible for the key in each successive hop.

A DHT's *routing structure* determines from which regions of identifier space a node chooses its neighbors. The

ideal routing structure is both flexible and scalable. With a flexible routing structure, a node is able to expand and contract the size of the routing table along a continuum in response to churn and bandwidth budget. With a scalable routing structure, even a very small routing table can lead to efficient lookups in a few hops. However, as currently defined, most DHT routing structures are scalable but not flexible and constrain which routing table sizes are possible. For example, a Tapestry node with a 160-bit identifier of base b maintains a routing table with $\frac{160}{\log_2 b}$ levels, each of which contain $b - 1$ entries. In practice, few of these levels are filled, and the expected number of neighbors per node in a network of n DHT nodes is $(b-1)\log_b n$. The parameter base (b) controls the table size, but it can only take values that are powers of 2, making it difficult to adjust the table size smoothly.

Existing routing structures are rigid in the sense that they *require* neighbors from certain regions of ID space to be present in the routing table. We can relax the table structure by specifying only the distribution of ID space distances between a node and its neighbors. Viewing routing structure as a probabilistic distribution gives a node the flexibility to use a routing table of any size. We model the distribution after proposed scalable routing structures. The ID space is organized as a ring as in Chord [25] and we define the ID distance to be the clockwise distance between two nodes on the ring.

Accordion uses a $\frac{1}{x}$ distribution to choose its neighbors: the probability of a node selecting a neighbor with distance x from itself in the identifier space from itself is proportional to $\frac{1}{x}$. This distribution causes a node to prefer neighbors that are closer to itself in ID space, ensuring that as a lookup gets closer to the target key there is always likely to be a helpful routing table entry. This $\frac{1}{x}$ distribution is the same as the "small-world" model proposed by Kleinberg [13], previously used by DHTs such as Symphony [19] and Mercury [1]. The $\frac{1}{x}$ distribution is also scalable and results in $O(\frac{\log n \log\log n}{\log s})$ lookup hops if each node has a table size of s; this result follows from an extension of Kleinberg's analysis [13].

3.2 Routing State Acquisition

A straightforward approach to learning new neighbors is to search actively for nodes with the $\frac{1}{x}$ distribution. A more bandwidth-efficient approach, however, is to learn about new neighbors, and the liveness of existing neighbors, as a side-effect of ordinary lookup traffic.

Learning through lookups does not necessarily yield useful information about existing neighbors or about new neighbors with the desired distribution in ID space. For example, if the DHT used *iterative routing* [25] during lookups, the original querying node would talk directly to each hop of the lookup. Assuming the keys being looked up are uniformly distributed, the querying node would communicate with nodes in a uniform distribution rather than a $\frac{1}{x}$ distribution.

With *recursive routing*, on the other hand, intermediate hops of a lookup forward the lookup message directly to the next hop. This means that nodes communicate only with existing neighbors from their routing tables during lookups. If each hop of a recursive lookup is acknowledged, then a node can check the liveness of a neighbor with each lookup it forwards, and the neighbor can piggyback information about its own neighbors in the acknowledgment.

If lookup keys are uniformly distributed and the nodes already have routing tables following a small-world distribution, then each lookup will involve one hop at exponentially smaller intervals in identifier space. Therefore, a node forwards lookups to next-hop nodes that fit its small-world distribution. A node can then learn about entries immediately following the next-hop nodes in identifier space, ensuring that the new neighbors learned also follow this distribution.

In practice lookup keys are not necessarily uniformly distributed, and thus Accordion devotes a small amount of its bandwidth budget to actively exploring for new neighbors according to the small-world distribution.

A DHT can learn even more from lookups if it performs parallel lookups, by sending out multiple copies of each lookup down different lookup paths. This increases the opportunity to learn new information, while at the same time decreasing lookup latency by circumventing potential timeouts. Analysis of DHT design techniques show that learning extra information from parallel lookups is more efficient at lowering lookup latencies than checking existing neighbor liveness or active exploration [16]. Accordion adjusts the degree of lookup parallelism based on the current lookup load to stay within the specified bandwidth budget.

3.3 Routing State Freshness

A DHT node must strike a balance between the freshness and the size of its routing table. While parallel lookups can help mask timeouts caused by stale entries, nodes still need to judge the freshness of entries to decide when to evict nodes, in order to limit the number of expected lookup timeouts.

Timeouts are expensive as nodes need to wait multiple round trip times to declare the lookup message failed before re-issuing it to a different neighbor [2, 22]. In order to avoid timeouts, most existing DHTs [2, 5, 20, 26] contact each neighbor periodically to determine the routing entry's liveness. In other words, a node can control its routing state freshness by evicting neighbors from its routing table that it has not successfully contacted for some interval. If the bandwidth budget were infinite, the node could ping each neighbor often to maintain fresh tables of arbitrarily large

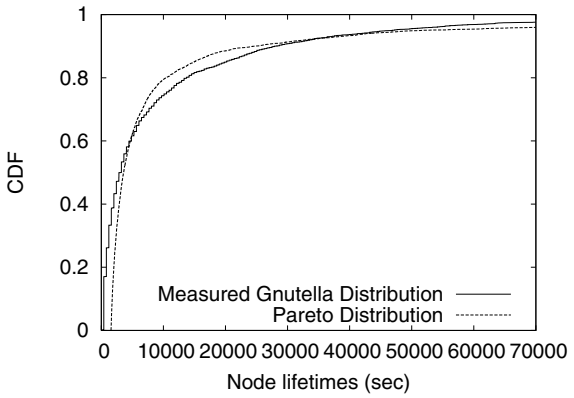


Figure 1: Cumulative distribution of measured Gnutella node uptime [24] compared with a Pareto distribution using $\alpha = 0.83$ and $\beta = 1560$ sec.

size. However, with a finite bandwidth, a DHT node must somehow make a tradeoff between the freshness and the size of its routing table. This section describes how to predict the freshness of routing table entries so that entries can be evicted efficiently.

3.3.1 Characterizing Freshness

The freshness of a routing table entry can be characterized with p, the probability of a neighbor being alive. The eviction process deletes a neighbor from the table if the estimated probability of it being alive is below some threshold p_{thresh}. Therefore, we are interested in finding a value for p_{thresh} such that the total number of lookup hops including timeout retries are minimized. If node lifetimes follow a memoryless exponential distribution, p is determined only by Δt_{since}, where Δt_{since} is the time interval since the neighbor was last known to be alive. However, in real systems, the distribution of node lifetimes is often heavy-tailed: nodes that have been alive for a long time are more likely to stay alive for an even longer time. In a heavy-tailed Pareto distribution, for example, the probability of a node dying before time t is

$$\Pr(\textit{lifetime} < t) = 1 - \left(\frac{\beta}{t}\right)^{\alpha}$$

where α and β are the shape and scale parameters of the distribution, respectively. Saroiu et al. measure such a distribution in a study of the Gnutella network [24]; in Figure 1 we compare their measured Gnutella lifetime distribution with a synthetic heavy-tailed Pareto distribution (using $\alpha = .83$ and $\beta = 1560$ sec). In a heavy-tailed distribution, p is determined by both the time when the node joined the network, Δt_{alive}, and Δt_{since}. We will present our estimation of p_{thresh} assuming a Pareto distribution for node lifetimes.

Let Δt_{alive} be the time for which the neighbor has been a member of the DHT, measured at the time it was last heard, Δt_{since} seconds ago. The conditional probability of a neighbor being alive, given that it had already been alive for Δt_{alive} seconds, is

$$p = \Pr(\textit{lifetime} > (\Delta t_{alive} + \Delta t_{since}) \mid \textit{lifetime} > \Delta t_{alive})$$

$$= \frac{(\frac{\beta}{\Delta t_{alive}+\Delta t_{since}})^{\alpha}}{(\frac{\beta}{\Delta t_{alive}})^{\alpha}} = \left(\frac{\Delta t_{alive}}{\Delta t_{alive} + \Delta t_{since}}\right)^{\alpha} \quad (1)$$

Therefore, $\Delta t_{since} = \Delta t_{alive}(p^{-\frac{1}{\alpha}} - 1)$. Since Δt_{alive} follows a Pareto distribution, the median lifetime is $2^{\frac{1}{\alpha}}\beta$. Therefore, within $\Delta t_{since} = 2^{\frac{1}{\alpha}}\beta(p_{thresh}^{-\frac{1}{\alpha}} - 1)$ seconds, half of the routing table should be evicted with the eviction threshold set at p_{thresh}. If s_{tot} is the total routing table size, the eviction rate is approximately $\frac{s_{tot}}{2\Delta t_{since}}$.

Since nodes aim to keep their maintenance traffic below a certain bandwidth budget, they can only refresh or learn about new neighbors at some finite rate determined the budget. For example, if a node's bandwidth budget is 20 bytes per second, and learning liveness information for a single neighbor costs 4 bytes (*e.g.*, the neighbor's IP address), then at most a node could refresh or learn routing table entries for 5 nodes per second.

Suppose that a node has a bandwidth budget such that it can afford to refresh/learn about B nodes per second. The routing table size s_{tot} at the equilibrium between eviction and learning is:

$$\frac{s_{tot}}{2\Delta t_{since}} = B$$

$$\Rightarrow s_{tot} = 2B\Delta t_{since} = 2B(2^{\frac{1}{\alpha}})\beta(p_{thresh}^{-\frac{1}{\alpha}} - 1) \quad (2)$$

However, some fraction of the table points to dead neighbors and therefore does not contribute to lowering lookup hops. The *effective* routing table size, then, is $s = s_{tot} \cdot p_{thresh}$.

3.3.2 Choosing the Best Eviction Threshold

Our goal is to choose a p_{thresh} that will minimize the expected number of hops for each lookup. We know from Section 3.1 that the average number of hops per lookup in a static network is $O(\frac{\log n \log\log n}{\log s})$; under churn, however, each hop successfully taken has an extra cost associated with it, due to the possibility of forwarding lookups to dead neighbors. When each neighbor is alive with probability at least p_{thresh}, the upper bound on the expected number of trials per successful hop taken is $\frac{1}{p_{thresh}}$ (for now, we assume no parallelism). Thus, we can approximate the expected number of actual hops per lookup, h, by multiplying the number

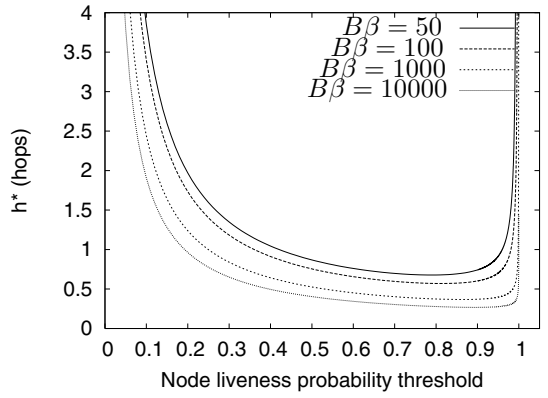


Figure 2: The function h^* (Equation 4) with respect to p_{thresh}, for different values of $B\beta$ and fixed $\alpha = 1$. h^* goes to infinity as p_{thresh} approaches 1.

of effective lookup hops with the expected number of trials needed per effective hop:

$$h \propto \frac{\log n \log\log n}{\log s} \cdot \frac{1}{p_{thresh}}$$

We then substitute the effective table size s with $s_{tot} \cdot p_{thresh}$, using Equation 2:

$$h \propto \frac{\log n \log\log n}{\log(2B\beta(2^{\frac{1}{\alpha}})(p_{thresh}^{-\frac{1}{\alpha}} - 1) \cdot p_{thresh})} \cdot \frac{1}{p_{thresh}} \quad (3)$$

The numerator of Equation 3 is constant with respect to p_{thresh}, and therefore can be ignored for the purposes of minimization. It usually takes on the order of a few round-trip times to detect lookup timeout and this multiplicative timeout penalty can also be ignored. Our task now is to choose a p_{thresh} that will minimize:

$$h^* = \frac{1}{\log(2B\beta(2^{\frac{1}{\alpha}})(p_{thresh}^{-\frac{1}{\alpha}} - 1)p_{thresh}) \cdot p_{thresh}} \quad (4)$$

The minimizing p_{thresh} depends on the constants $(B\beta) \cdot (2^{\frac{1}{\alpha}})$ and α. If p_{thresh} varied widely given different values of $B\beta$ and α, nodes would constantly need to reassess their estimates of p_{thresh} using rough estimates of the current churn rate and the bandwidth budget. Fortunately, this is not the case.

Figure 2 plots h^* with respect to p_{thresh}, for various values of $B\beta$ and a fixed α. We consider only values of $B\beta$ large enough to allow nodes to maintain a reasonable number of neighbors under the given churn rate. For example, if nodes have mean lifetimes of 10 seconds ($\beta = 5$ sec, $\alpha = 1$), but can afford to refresh/learn one neighbor per second, no value of p_{thresh} will allow s to be greater than 2.

Figure 2 shows that as p_{thresh} increases the expected lookup hops decreases due to fewer timeouts; however, as p_{thresh} becomes even larger and approaches 1, the number of hops actually increases due to a limited table size. The p_{thresh} that minimizes lookup hops lies somewhere between .7 and .9 for all curves. Figure 2 also shows that as $B\beta$ increases, the p_{thresh} that minimizes h^* increases as well, but only slightly. In fact, for any reasonable value of $B\beta$, h^* varies so little around its true minimum that we can approximate the optimal p_{thresh} for any value of $B\beta$ to be .9. A similar analysis shows the same results for reasonable α values. For the remainder of this paper, we assume $p_{thresh} = .9$, because even though this may not be precisely optimal, it will produce an expected number of hops that is nearly minimal in most deployment scenarios.

The above analysis for p_{thresh} assumes no lookup parallelism. If lookups are sent down multiple paths concurrently, nodes can use a much smaller value for p_{thresh} because the probability will be small that *all* of the next-hop messages will timeout. Using a smaller value for p_{thresh} leads to a larger effective routing table size, reducing the average lookup hop count. Nodes can choose a p_{thresh} value such that the probability that at least one next-hop message will not fail is at least .9.

3.3.3 Calculating Entry Freshness

Nodes can use Equation 1 to calculate p, the probability of a neighbor being alive, and then evict entries with $p < p_{thresh}$. Calculating p requires estimates of three values: Δt_{alive} and Δt_{since} for the given neighbor, along with the shape parameter α of the Pareto distribution. Interestingly, p does not depend on the scale parameter β, which determines the median node lifetime in the system. This is counterintuitive; we expect that smaller median node lifetimes (*i.e.*, faster churn rates) will decrease p and increase the eviction rate. This median lifetime information, however, is implicitly present in the observed values for Δt_{alive} and Δt_{since}, so β is not explicitly required to calculate p.

Equation 1, as stated, still requires some estimate for α, which may be difficult to observe and calculate. To simplify this task, we define an indicator variable i for each routing table entry as follows:

$$i = \frac{\Delta t_{alive}}{\Delta t_{alive} + \Delta t_{since}} \quad (5)$$

Since $p = i^\alpha$, a monotonically increasing function of i, there exists some i_{thresh} such that any routing table entry with $i < i_{thresh}$ will also have a $p < p_{thresh}$. Thus, if nodes can estimate the value of i_{thresh} corresponding to p_{thresh}, no estimate of α is necessary. All entries with i less than i_{thresh} will be evicted. Section 4.6 describes how Accordion estimates an appropriate i_{thresh} for the observed churn, and how nodes learn Δt_{alive} and Δt_{since} for each entry.

4 The Accordion Protocol

Accordion uses consistent hashing [12] in a circular identifier space to assign keys to nodes. Accordion borrows Chord's protocols for maintaining a linked list from each node to the ones immediately following in ID space (Chord's successor lists and join protocol). An Accordion node's routing table consists of a set of neighbor entries, each containing a neighboring node's IP address and ID.

An Accordion lookup for a key finds the key's successor: the node whose ID most closely follows the key in ID space. When node n_0 starts a query for key k, n_0 looks in its routing table for the neighbor n_1 whose ID most closely precedes k, and sends a query packet to n_1. That node follows the same rule: it forwards the query to the neighbor n_2 that most closely precedes k. When the query reaches node n_i and k lies between n_i and the n_i's successor, the query has finished; n_i sends a reply directly back to n_0 with the identity of its successor (the node responsible for k).

4.1 Bandwidth Budget

Accordion's strategy for using the bandwidth budget is to use as much bandwidth as possible on lookups by exploring multiple paths in parallel [16]. When some bandwidth is left over (perhaps due to bursty lookup traffic), Accordion uses the rest to explore; that is, to find new routing entries according to a small-world distribution.

This approach works well because parallel lookups serve two functions. Parallelism reduces the impact of timeouts on lookup latency because one copy of the lookup may proceed while other copies wait in timeout. Parallel lookups also allow nodes to learn about new nodes and about the liveness of existing neighbors, and as such it is better to learn as a side-effect of lookups than from explicit probing. Section 4.3 explains how Accordion controls the degree of lookup parallelism to try to fill the whole budget.

Accordion must also keep track of how much of the budget is left over and available for exploration. To control the budget, each node maintains an integer variable, b_{avail}, which keeps track of the number of bytes available to the node for exploration traffic, based on recent activity. Each time the node sends a packet or receives the corresponding acknowledgment (for any type of traffic), it decrements b_{avail} by the size of the packet. It does not decrement b_{avail} for unsolicited incoming traffic, or for the corresponding outgoing acknowledgments. In other words, each packet only counts towards the bandwidth budget at one end. Periodically, the node increments b_{avail} at the rate of the bandwidth budget.

The user gives the bandwidth budget in two parts: the average desired rate of traffic in bytes per second (r_{avg}), and the maximum burst size in bytes (b_{burst}). Every t_{inc} seconds, the node increments b_{avail} by $r_{avg} \cdot t_{inc}$ (where t_{inc} is the size of one exploration packet divided by r_{avg}). Whenever b_{avail} is positive, the node sends one exploration packet, according to the algorithm we present in Section 4.4. Nodes decrement b_{avail} down to a minimum of $-b_{burst}$. While $b_{avail} = -b_{burst}$, nodes immediately stop sending all low priority traffic (such as redundant lookup traffic and exploration traffic). Thus, nodes send no exploration traffic unless the average traffic over the last b_{burst}/r_{avg} seconds has been less than r_{avg}.

The bandwidth budget controls the maintenance traffic sent by an Accordion node, but does not give the node direct control over all incoming and outgoing traffic. For example, a node must acknowledge all traffic sent to it from its predecessor regardless of the value of b_{avail}; otherwise, its predecessor may think it has failed and the correctness of lookups would be compromised. The imbalance between a node's specified budget and its actual incoming and outgoing traffic is of special concern in scenarios where nodes have heterogeneous budgets in the system. To help nodes with low budgets avoid excessive incoming traffic from nodes with high budgets, an Accordion node biases lookup and table exploration traffic toward neighbors with higher budgets. Section 4.5 describes the details of this bias.

4.2 Learning from Lookups

When an Accordion node forwards a lookup (see Figure 4.2), the immediate next-hop node returns an acknowledgment that includes a set of neighbors from its routing table; this acknowledgment allows nodes to learn from lookups. The acknowledgment also serves to indicate that the next-hop is alive.

If n_1 forwards a lookup for key k to n_2, n_2 returns a set of neighbors in the ID range between n_2 and k. Acquiring new entries this way allow nodes to preferentially learn about ID spaces close-by to itself, the key characteristic of a small-world distribution. Additionally, the fact that n_1 forwarded the lookup to n_2 indicates that n_1 does not know of any nodes in the ID gap between n_2 and k, and n_2 is well-situated to fill this gap.

4.3 Parallel Lookups

An Accordion node increases the parallelism of lookups it initiates and forwards until the point where the lookup traffic nearly fills the bandwidth budget. An Accordion node must adapt the level of parallelism as the underlying lookup rate changes, it must avoid forwarding the same lookup twice, and it must choose the most effective set of nodes to which to forward copies of each lookup.

A key challenge in Accordion's parallel lookup design is caused by its use of recursive routing. Previous DHTs with parallel lookups use iterative routing: the originating node sends lookup messages to each hop of the lookup in

```
procedure NEXTHOP(lookup_request q)
 if this node owns q.key  then {
      reply to lookup source directly
      return (NULL)
 }
 // use bias to pick best predecessor (Section 4.5)
 nexthop ← routetable.BESTPRED(q.key)
 // forward query to next hop
 // and wait for ACK and learning info
 nextreply ← nexthop.NEXTHOP(q)
 put nodes of nextreply in routetable
 // find some nodes between this node
 // and the key, and return them
 return (GETNODES(q.lasthop, q.key))

procedure GETNODES(src, end)
 s ← neighbors between me and end
 // m is some constant (e.g., 5)
 if s.SIZE() < m  then v ← s
 else v ← m nodes in s nearest to src w.r.t. latency
 return (v)
```

Figure 3: Learning from lookups in Accordion.

turn [15,20]. Iterative lookups allow the originating node to explicitly control the amount of parallelism and the order in which paths are explored, since the originating node issues all messages related to the lookup. However, Accordion uses recursive routing to learn nodes with a small-world distribution, and nodes forward lookups directly to the next hop. To control recursive parallel lookups, each Accordion node independently adjusts its lookup parallelism to stay within the bandwidth budget.

If an Accordion node knew the near-term future rate at which it was about to receive lookups to be forwarded, it could divide the bandwidth budget by that rate to determine the level of parallelism. Since it cannot predict the future, Accordion uses an adaptive algorithm to set the level of parallelism based on the past lookup rate. Each node maintains a w_p "parallelism window" variable that determines the number of copies it forwards of each received or initiated lookup. A node updates w_p every t_p seconds, where $t_p = b_{burst}/r_{avg}$, which allows enough time for the bandwidth budget to recover from potential bursts of lookup traffic. During each interval of t_p seconds, a node keeps track of how many unique lookup packets it has originated or forwarded, and how many exploration packets it has sent. If more exploration packets have been sent than the number of lookups that have passed through this node, w_p increases by 1. Otherwise, w_p decreases by half. This additive increase/multiplicative decrease (AIMD) style of control ensures a prompt response to w_p overestimation or sudden changes in the lookup load. Additionally, nodes do not increase w_p above some maximum value, as determined by the maximum burst size, b_{burst}. A node forwards the w_p copies of a lookup to the w_p neighbors whose IDs most closely precede the desired key in ID space.

When a node originates a query, it marks one of the parallel copies with a "primary" flag which gives that copy high priority. Intermediate nodes are free to drop non-primary copies of a query if they do not have sufficient bandwidth to forward the query, or if they have already seen a copy of the query in the recent past. If a node receives a primary query, it marks one forwarded copy as primary, maintaining the invariant that there is always one primary copy of a query. Primary lookup packets trace the path a non-parallel lookup would have taken, while non-primary traffic copies act as optional traffic to decrease timeout latency and increase information learned.

4.4 Routing Table Exploration

When lookup traffic is bursty, Accordion might not be able to accurately predict w_p for the next time period. As such, parallel lookups would not consume the entire bandwidth budget during that time period. Accordion uses this leftover bandwidth to explore for new neighbors actively. Because lookup keys are not necessarily distributed uniformly in practice, a node might not be able to learn new entries with the correct distribution through lookups alone; explicit exploration addresses this problem. The main goal of exploration is that it be bandwidth-efficient and result in learning nodes with the small-world distribution described in Section 3.1.

For each neighbor x ID-distance away from a node, the gap between that neighbor and the next successive entry should be proportional to x. A node with identifier a compares the *scaled gaps* between successive neighbors n_i and n_{i+1} to decide the portion of its routing table most in need of exploration. The scaled gap g between neighbors n_i and n_{i+1} is:

$$g = \frac{d(n_i, n_{i+1})}{d(a, n_i)}$$

where $d(x, y)$ computes the clockwise distance in the circular identifier space between identifiers x and y. When an Accordion node sends an exploration query, it sends it to the neighbor with the largest scaled gap between it and the next neighbor. The result is that the node explores in the area of ID space where its routing table is the most sparse with respect to the desired distribution.

An exploration message from node a asks neighbor n_i for m neighbor entries between n_i and n_{i+1}, where m is some small constant (*e.g.*, 5). n_i retrieves these entries from both its successor list and its routing table. n_i uses Vivaldi network coordinates [4] to find the m nodes in this gap with the lowest predicted network delay to a. If n_i returns fewer

than m entries, node a will not revisit n_i again until it has explored all other neighbors.

The above process only *approximates* a $\frac{1}{x}$ distribution; it does not guarantee such a distribution in all cases. Such a guarantee would not be flexible enough to allow a full routing table when bandwidth is plentiful and churn is low. Accordion's exploration method results in a $\frac{1}{x}$ distribution when churn is high, but also achieves nearly full routing tables when the bandwidth budget allows.

4.5 Biasing Traffic to High-Budget Nodes

Because nodes have no direct control over their incoming bandwidth, in a network containing nodes with diverse bandwidth budgets we expect that some nodes will be forced over-budget by incoming traffic from nodes with bigger budgets. Accordion addresses this budgetary imbalance by biasing lookup and exploration traffic toward nodes with higher budgets. Though nodes still do not have direct control over their incoming bandwidth, in the absence of malicious nodes this bias serves to distribute traffic in proportion to the bandwidth budgets of nodes.

When an Accordion node learns about a new neighbor, it also learns that neighbor's bandwidth budget. Traditional DHT protocols (*e.g.*, Chord) route lookups greedily to the neighbor most closely preceding the key in ID space, because that neighbor is expected to have the highest density of routing entries near the key. We generalize this idea to consider bandwidth budget. Since the density of routing entries near the desired ID region increases linearly with the node's bandwidth budget but decreases with the node's distance from that region in ID space, neighbors should forward lookup/exploration traffic to the neighbor with the best *combination* of high budget and short distance.

Suppose a node a decides to send an exploration packet to its neighbor n_1 (with budget b_1), to learn about new entries in the gap between n_1 and the following entry n_0 (as discussed in Section 4.4). Let x be the distance in identifier space between n_1 and the following entry n_0. Let n_i ($i = 2, 3...$) be neighbors preceding n_1 in the a's routing table, each with a bandwidth budget of b_i. In Accordion's traffic biasing scheme, a prefers to send the exploration packet to the neighbor n_i ($i = 1, 2...$) with the largest value for the following equation:

$$v_i = \frac{b_i}{d(n_i, n_1) + x}$$

where $x = d(n_1, n_0)$. In the case of making lookup forwarding decisions for some key k, $x = d(n_1, k)$ and n_1 is the entry immediately precedes k in a's routing table. For each lookup and exploration decision, an Accordion node examines a fixed number of candidate neighbors (set to 8 in our implementation) preceding n_1 and also ensures that the lookup progresses at least halfway towards the key if possible.

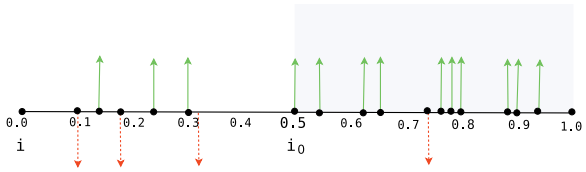


Figure 4: A list of contact entries, sorted by increasing i values. Up arrows indicate events where the neighbor was alive, and down arrows indicate the opposite. A node estimates i_0 to be the minimum i such that there are more than 90% (p_{thresh}) live contacts for $i > i_0$, and then incorporates i_0 into its i_{thresh} estimate.

To account for network proximity, Accordion further weights the v_i values by the estimated network delay to the neighbor based on network coordinates. With this extension, a chooses the neighbor with the largest value for $v'_i = v_i / delay(a, n_i)$. This is similar in spirit to traditional proximity routing schemes [7].

4.6 Estimating Liveness Probabilities

In order to avoid timeout delays during lookups, an Accordion node must ensure that the neighbors in its routing table are likely to be alive. Accordion does this by estimating each neighbor's probability of being alive, and evicting neighbors judged likely to be dead. For any reasonable node lifetime distribution, the probability that a node is alive decreases as the amount of time since the node was last heard from increases. Accordion attempts to calculate this probability explicitly.

Section 3.3 showed that for a Pareto node lifetime distribution, nodes should evict all entries whose probability of being alive is less than some threshold p_{thresh} so the probability of successfully forwarding a lookup is greater than .9 given the current lookup parallelism w_p (*i.e.*, $1 - (1 - p_{thresh})^{w_p} = 0.9$). The value i from Equation 5 indicates the probability p of a neighbor being alive. The overall goal of Accordion's node eviction policy is to estimate a value for i_{thresh}, such that nodes evict any neighbor with an associated i value below i_{thresh}. See Section 3.3 for the definitions of i and i_{thresh}.

A node estimates i_{thresh} as follows. Each time it contacts a neighbor, it records whether the neighbor is alive or dead and the neighbor's current indicator value i. Periodically, a node reassesses its estimation of i_{thresh} using this list. It first sorts all the entries in the list by increasing i value, and then determines the smallest value i_0 such that the fraction of entries with an "alive" status and an $i > i_0$ is p_{thresh}. The node then incorporates i_0 into its current estimate of i_{thresh}, using an exponentially-weighted moving average. Figure 4 shows the correct i_0 value for a given sorted list of entries.

To calculate i for each neighbor using Equation 5, nodes must know Δt_{alive} (the time between when the neighbor last joined the network and when it was last heard) and Δt_{since} (the time between when it was last heard and now). Each node keeps track of its own Δt_{alive} based on the time of its last join, and includes its own Δt_{alive} in every packet it sends. Nodes learn (Δt_{alive}, Δt_{since}) information associated with neighbors in one of the following three ways:

- When the node hears from a neighbor directly, it records the current local timestamp as t_{last} in the routing entry for that neighbor, and resets an associated Δt_{since} value to 0 and sets Δt_{alive} to the newly-received Δt_{alive} value.
- If a node hears information about a new neighbor indirectly from another node, it will save the supplied Δt_{since} value in the new routing entry, and set the entry's t_{last} value to the current local timestamp.
- If a node hears information about an existing neighbor, it compares the received Δt_{since} value with its currently recorded value for that neighbor. A smaller received Δt_{since} indicates fresher information about this neighbor, and so the node saves the corresponding (Δt_{alive}, Δt_{since}) pair for the neighbor in its routing table. It also sets t_{last} to the current local timestamp.

Whenever a node needs to calculate a current value for Δt_{since} (either to compare its freshness, to estimate i, or to pass it to a different node), it adds the saved Δt_{since} value and the difference between the current local timestamp and t_{last}.

5 Evaluation

This section demonstrates the important properties of Accordion through simulation. It shows that Accordion matches the performance of existing $\log n$-routing-table DHTs when bandwidth is scarce, and the performance of large-table DHTs when bandwidth is plentiful under different lookup workloads. Accordion achieves low latency lookups under varying network sizes and churn rates with bounded routing table maintenance overhead. Furthermore, Accordion's automatic self-tuning algorithms approach the best possible performance/cost tradeoff, and Accordion's performance degrades only modestly when the node lifetimes do not follow the assumed Pareto distribution. Accordion stays within its bandwidth budget on average even when nodes have heterogeneous bandwidth budgets.

5.1 Experimental Setup

This evaluation uses an implementation of Accordion in p2psim, a publicly-available, discrete-event packet level simulator. Existing p2psim implementations of the Chord and OneHop DHTs simplified comparing Accordion to these protocols. The Chord implementation chooses neighbors based on their proximity [5, 7].

For simulations involving networks of less than 1740 nodes, we use a pairwise latency matrix derived from measuring the inter-node latencies of 1740 DNS servers using the King method [8]. However, because of the limited size of this topology and the difficulty involved in obtaining a larger measurement set, for simulations involving larger networks we assign each node a random 2D synthetic Euclidean coordinate and derive the network delay between a pair of nodes from their corresponding Euclidean distance. The average round-trip delay between node pairs in both the synthetic and measured delay matrices is 179 ms. Since each lookup for a random key starts and terminates at two random nodes, the average inter-host latency of the topology serves as a lower bound for the average DHT lookup latency. By default, our experiments use a Euclidean topology of 3000 nodes, except when noted. p2psim does not simulate link transmission rates or queuing delays. The experiments involve only key lookups; no data is retrieved.

Each node alternately leaves and re-joins the network; the interval between successive events for each node follows a Pareto distribution with median time of 1 hour (*i.e.*, $\alpha = 1$ and $\beta = 1800$ sec), unless noted. This choice of lifetime distribution is similar to past studies of peer-to-peer networks, as discussed in Section 3.3. Because $\alpha = 1$ in all simulations involving a Pareto distribution, our implementation of Accordion does not use the i_{thresh}-estimation technique presented in Section 4.6, as it is more convenient to set $i_{thresh} = p_{thresh} = .9$ instead.

Nodes issue lookups with respect to two different workloads. In the *churn intensive* workload, each node issues a lookup once every 10 minutes, while in the *lookup intensive* workload, each node issues a lookup once every 9 seconds. Experiments use the churn intensive workload unless otherwise noted. Each time a node joins, it uses a different IP address and DHT identifier. Each experiment runs for four hours of simulated time; statistics are collected only during the final half of the experiment and averaged over 5 simulation runs. All Accordion configurations set $b_{burst} = 100 r_{avg}$.

5.2 Comparison Framework

We evaluate the performance of the protocols using two types of metrics, *performance* and *cost*, following from the performance versus cost framework (PVC) we developed in previous work [16]. Though other techniques exist for comparing DHTs under churn [14, 17], PVC naturally allows us to measure how efficiently protocols achieve their performance vs. cost tradeoffs.

We measure performance as the average lookup latency

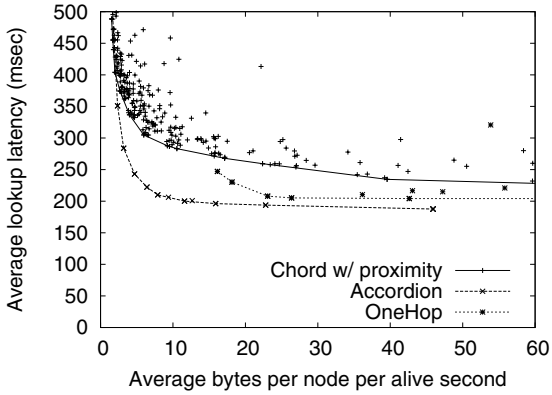


Figure 5: Accordion's bandwidth vs. lookup latency tradeoff compared to Chord and OneHop, using a 3000-node network and a churn intensive workload. Each point represents a particular parameter combination for the given protocol. Accordion's performance matches or improves OneHop's when bandwidth is plentiful, and Chord's when bandwidth is constrained.

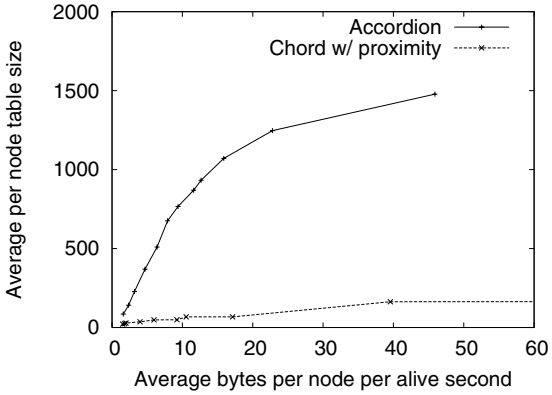


Figure 6: The average routing table size for Chord and Accordion as a function of the average per-node bandwidth, using a 3000-node network and a churn intensive workload. The routing table sizes for Chord correspond to the optimal parameter combinations in Figure 5. Accordion's ability to grow its routing table as available bandwidth increases explains why its latency is generally lower than Chord's.

of correct lookups (*i.e.*, lookups for which a correct answer is returned), including timeout penalties (three times the round-trip time to the dead node). All protocols retry failed lookups (*i.e.*, lookups that time out without completing) for up to a maximum of four seconds. We do not include the latencies of incorrect or failed lookups in this metric, but for all experiments of interest these counted for less than 5% of the total lookups for all protocols.

We measure cost as the average bandwidth consumed per node per alive second (*i.e.*, we divide the total bytes consumed by the sum of times that each node was alive). The size in bytes of each message is counted as 20 bytes for headers plus 4 bytes for each node mentioned in the message for Chord and OneHop. Each Accordion node entry is counted as 8 bytes due to additional fields on the bandwidth budget, node membership time (Δt_{alive}), and time since last contacted (Δt_{since}) for each node entry.

For graphs comparing DHTs with many parameters (*i.e.*, Chord and OneHop) to Accordion, we use PVC to explore the parameter space of Chord and OneHop fully and scatterplot the results. Each point on such a figure shows the average lookup latency and bandwidth overhead measured for one distinct set of parameter values for those protocols. The graphs also have the *convex hull* segments of the protocols, which show the best latency/bandwidth tradeoffs possible with the protocols, given the many different configurations possible. Accordion, on the other hand, has only one parameter, the bandwidth budget, and does not need to be explored in this manner.

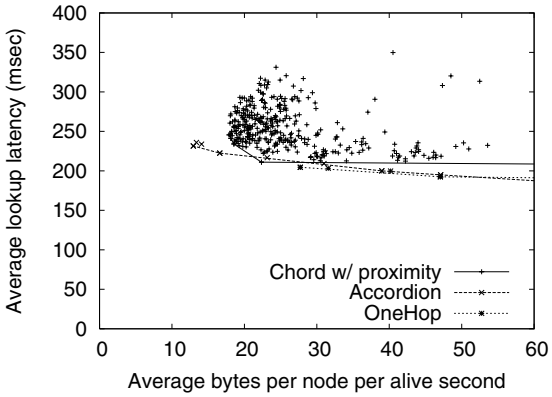


Figure 7: Accordion's lookup latency vs. bandwidth overhead tradeoff compared to Chord and OneHop, using a 1024-node network and a lookup intensive workload.

5.3 Latency vs. Bandwidth Tradeoff

A primary goal of the Accordion design is to adapt the routing table size to achieve the lowest latency depending on bandwidth budget and churn. Figure 5 plots the average lookup latency vs. bandwidth overhead tradeoffs of Accordion, Chord, and OneHop. In this experiment, we varied Accordion's r_{avg} parameter between 3 and 60 bytes per second. We plot measured actual bandwidth consumption, not the configured bandwidth budget, along the x-axis. The x-axis values include all traffic: lookups as well as routing table maintenance overhead.

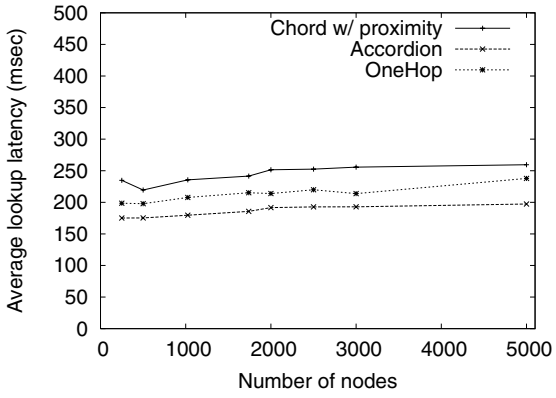


Figure 8: The lookup latency of Chord, Accordion and OneHop as the number of nodes in the system increases, using a churn intensive workload. Accordion uses a bandwidth budget of 6 bytes/sec, and the parameters of Chord and OneHop are fixed to values that minimize lookup latency when consuming 7 and 23 bytes/node/sec in a 3000-node network, respectively.

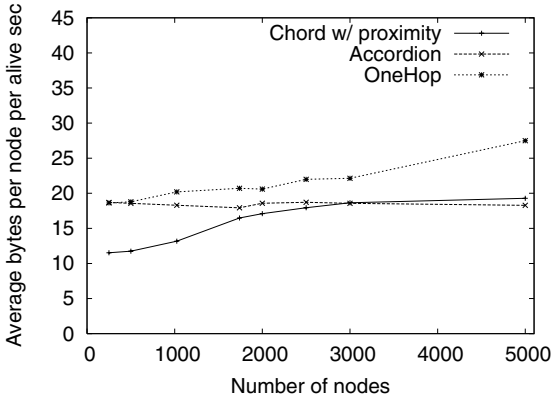


Figure 9: The average bytes consumed per node by Chord, Accordion and OneHop as the number of nodes in the system increases, from the same set of experiments as Figure 8.

Accordion approximates the lookup latency of the best OneHop configuration when the bandwidth budget is large, and the latency of the best Chord configuration when bandwidth is small. This is a result of Accordion's ability to adapt its routing table size, as illustrated in Figure 6. On the left, when the budget is limited, Accordion's table size is almost as small as Chord's. As the budgets grows, Accordion's routing table also grows, approaching the number of live nodes in the system (on average, half of the 3000 nodes are alive in the system).

As the protocols use more bandwidth, Chord cannot increase its routing table size as quickly as Accordion, even when optimally-tuned; instead, a node spends bandwidth on maintenance costs for its slowly-growing table. By increasing the table size more quickly, Accordion reduces the number of hops per lookup, and thus the average lookup latency.

Because OneHop keeps a complete routing table, all arrival and departure events must be propagated to all nodes in the system. This restriction prevents OneHop from being configured to consume very small amounts of bandwidth. As OneHop propagates these events more quickly, the routing tables are more up-to-date and both the expected hop count and timeouts per lookups decrease. Accordion, on the other hand, adapts its table size smoothly as its bandwidth budget allows, and can consistently maintain a fresher routing table, and thus lower latency lookups, than OneHop.

5.4 Effect of a Different Workload

The simulations in the previous section featured a workload that was *churn intensive*; that is, the amount of churn in the network was high in proportion to the lookup rate. This section evaluates the performance of Chord, OneHop, and Accordion under a *lookup intensive* workload. In this workload, each node issues one lookup every 9 seconds (almost 70 times more often than in the churn intensive workload), while the rate of churn is the same as that in the previous section.

Figure 7 shows the performance results for the three protocols. Again, convex hull segments and scatter plots characterize the performance of Chord and OneHop, while Accordion's latency/bandwidth curve is derived by varying the per-node bandwidth budget. As before, Accordion's performance approximates OneHop's when bandwidth is high.

In contrast to the churn intensive workload, in the lookup intensive workload Accordion can operate at lower levels of bandwidth consumption than Chord. With a low lookup rate as in Figure 5, Chord can be configured with a small base (and thus small routing table and more lookup hops, accordingly) to achieve low latencies, with relatively high lookup latencies. However, with a high lookup rate as in Figure 7, using a small base in Chord is not the best configuration: it has relatively high lookup latency, but also has a large overhead due to the large number of forwarded lookups. Because Accordion learns new routing entries from lookup traffic, a higher rate of lookups leads to a larger per-node routing table, resulting in fewer lookup hops and less overhead due to forwarding lookups. Thus, Accordion can operate at lower levels of bandwidth than Chord because it automatically increases its routing table size by learning from the large number of lookups.

The rest of the evaluation focuses on the churn intensive workload, unless otherwise specified.

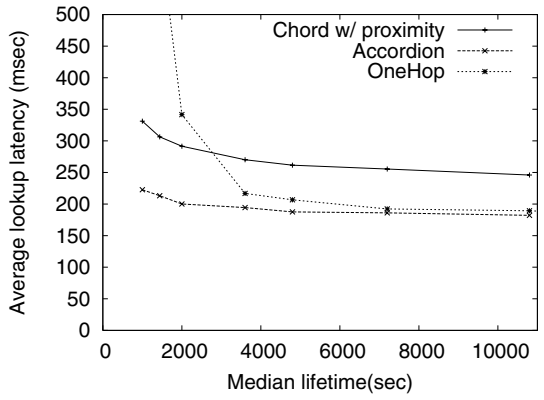


Figure 10: The lookup latency of Chord, Accordion and OneHop as median node lifetime increases (and churn decreases), using a 3000-node network. Accordion uses a bandwidth budget of 24 bytes/sec, and the parameters of Chord and OneHop are fixed to values that minimize lookup latency when consuming 17 and 23 bytes/node/sec, respectively, with median lifetimes of 3600 sec.

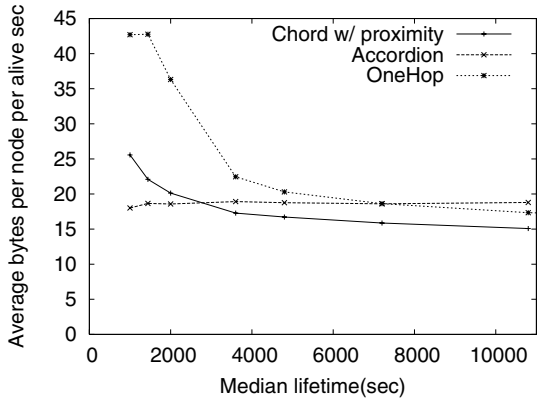


Figure 11: The average bytes consumed per node by Chord, Accordion and OneHop as median node lifetime increases (and churn decreases), from the same set of experiments as Figure 10.

5.5 Effect of Network Size

This section investigates the effect of scaling the size of the network on the performance of Accordion. Figures 8 and 9 show the average lookup latency and bandwidth consumption of Chord, Accordion and OneHop as a function of the network size. For Chord and OneHop, we fix the protocol parameters to be the optimal settings in a 3000-node network (*i.e.*, the parameter combinations that produce latency/overhead points lying on the convex hull segments) for bandwidth consumptions of 17 bytes/node/sec and 23 bytes/node/sec, respectively. For Accordion, we fix the bandwidth budget at 24 bytes/sec. With fixed parameter settings, Figure 9 shows that both Chord and OneHop incur increasing overhead that scales as $\log n$ and n respectively, where n is the size of the network. However, Accordion's fixed bandwidth budget results in predictable overhead consumption regardless of the network size. Despite using less bandwidth than OneHop and the fact that Chord's bandwidth consumption approaches that of Accordion as the network grows, Accordion's average lookup latency is consistently lower than that of both Chord and OneHop.

These figures plot the *average* bandwidth consumed by the protocols, which hides the bandwidth that is consumed on per-node or burst levels. Because Accordion controls bandwidth bursts, it keeps individual nodes within their bandwidth budgets. OneHop, however, explicitly distributes bandwidth unevenly: slice leaders [9] typically use 7 to 10 times the bandwidth of average nodes. OneHop is also more bursty than Accordion; we observe that the maximum bandwidth burst observed for OneHop is 1200 bytes/node/sec in a 3000-node network, more than 10 times the maximum burst of Accordion. Thus, OneHop's bandwidth consumption varies widely and could at any one time exceed a node's desired bandwidth budget, while Accordion stays closer to its average bandwidth consumption.

5.6 Effect of Churn

Previous sections illustrated Accordion's ability to adapt to different bandwidth budgets and network sizes; this section evaluates its adaptability to different levels of churn.

Figures 10 and 11 shows the lookup latency and bandwidth overhead of Chord, Accordion and OneHop as a function of median node lifetime. Lower node lifetimes correspond to higher churn. Accordion's bandwidth budget is constant at 24 bytes per second per node. Chord and OneHop uses parameters that achieve the lowest lookup latency while consuming 17 and 23 bytes per second, respectively, for a median node lifetime of one hour. While Accordion maintains fixed bandwidth consumption regardless of churn, both Chord and OneHop's overhead grow inversely proportional to median node lifetime (proportional to churn rates). Accordion's average lookup latency increases with shorter median node lifetimes, as it maintains a smaller table due to higher eviction rates under high churn. Chord's lookup latency increases due to a larger number of lookup timeouts, because of its fixed table stabilization interval. Accordion's lookup latency decreases slightly as the network becomes more stable, with consistently lower latencies than both Chord and OneHop. OneHop has unusually high lookup latencies under high churn as its optimal setting for the event aggregation interval with mean node lifetimes of 1 hour is not ideal under higher churn, and as a result lookups incur more frequent timeouts due to stale routing table entries.

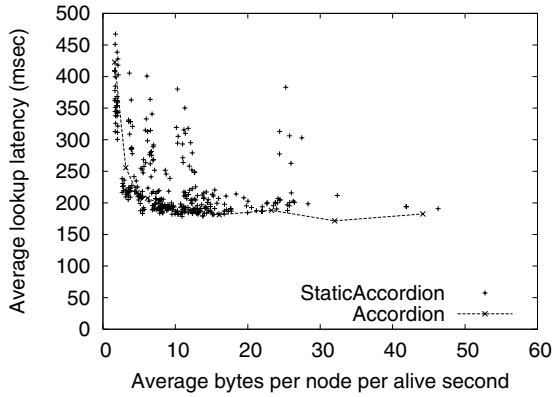


Figure 12: Bandwidth versus latency for Accordion and StaticAccordion, using a 1024-node network and a churn intensive workload. Accordion tunes itself nearly as well as the best exhaustive-search parameter choices for StaticAccordion.

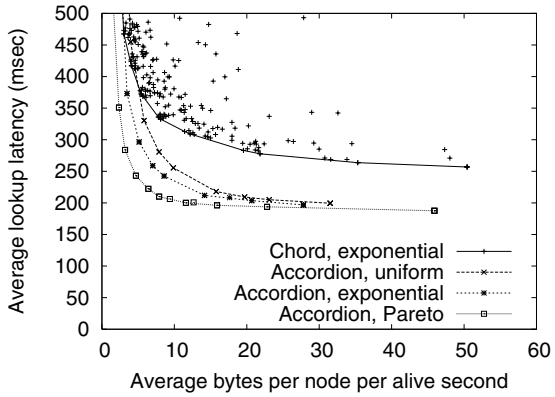


Figure 13: The performance of Accordion on three different node lifetime distributions, and of Chord on an exponential distribution, using a 3000-node network and a churn intensive workload. Though Accordion works best with a Pareto distribution, it still outperforms Chord with an exponential node lifetime distribution in most cases.

Parameter	**Range**
Exploration interval	2-90 sec
Lookup parallelism w_p	1,2,4,6
Eviction threshold i_{thresh}	.6 –.99

Table 1: StaticAccordion parameters and ranges.

5.7 Effectiveness of Self-Tuning

Accordion adapts to the current churn and lookup rate by adjusting w_p and the frequency of exploration, in order to stay within its bandwidth budget. To evaluate the quality of the adjustment algorithms, we compare Accordion with a simplified version (called StaticAccordion) that uses fixed w_p, i_{thresh} and active exploration interval parameters. Simulating StaticAccordion with a range of parameters, and looking for the best latency vs. bandwidth tradeoffs, indicates how well Accordion could perform with ideal parameter settings. Table 1 summarizes StaticAccordion's parameters and the ranges explored.

Figure 12 plots the latency vs. bandwidth tradeoffs of StaticAccordion for various parameter combinations. The churn and lookup rates are the same as the scenario in Figure 5. The lowest StaticAccordion points, and those farthest to the left, represent the performance Accordion could achieve if it self-tuned its parameters optimally. Accordion approaches the best static tradeoff points, but has higher latencies in general for the same bandwidth consumption. This is because Accordion tries to control bandwidth overhead, such that it not exceed the maximum-allowed burst size if possible (where we let $b_{burst} = 100 r_{avg}$). StaticAccordion, on the other hand, does not attempt to regulate its burst size. For example, when the level of lookup parallelism is high, a burst of lookups will generate a large burst of traffic. However, Accordion will reduce the lookup parallelism w_p to try to stay with the maximum burst size. Therefore, StaticAccordion can keep its lookup parallelism constant to achieve lower latencies (by masking more timeouts) than Accordion, though the average bandwidth consumption will be the same in both cases. As such, if controlling bursty bandwidth is a goal of the DHT application developer, Accordion will control node bandwidth more consistently than StaticAccordion, without significant additional lookup latency.

5.8 Lifetime Distribution Assumption

Accordion's algorithm for predicting neighbor liveness probability assumes a heavy-tailed Pareto distribution of node lifetimes (see Sections 3.3 and 4.6). In such a distribution, nodes that have been alive a long time are likely to remain alive. Accordion exploits this property by preferring to keep long-lived nodes in the routing table. If the distribution of lifetimes is not what Accordion expects, it may make more mistakes about which nodes to keep, and thus suffer more lookup timeouts. This section evaluates the effect of such mistakes on lookup latency.

Figure 13 shows the latency/bandwidth tradeoff with node lifetime distributions that are uniform and exponential. The uniform distribution chooses lifetimes uniformly at random between six minutes and nearly two hours, with an average of one hour. In this distribution, nodes that have been part of the network longer are *more* likely to fail soon. In the exponential distribution, node lifetimes are exponentially distributed with a mean of one hour; the probability of a node being alive does not depend on its join time.

Figure 13 shows that Accordion's lookup latencies are

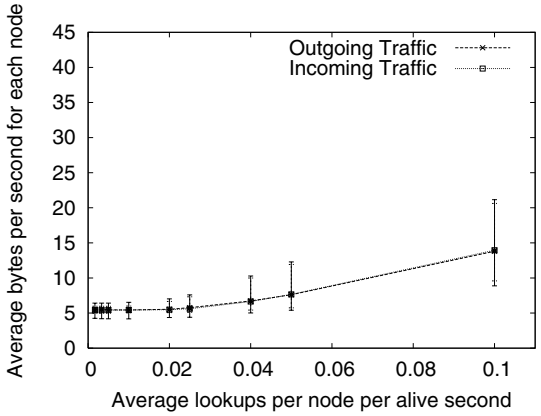


Figure 14: Accordion's bandwidth consumption vs. lookup rate, using a 3000-node network and median node lifetimes of one hour. All nodes have a bandwidth budget of 6 bytes/sec. Nodes stay within the budget until the lookup traffic exceeds that budget.

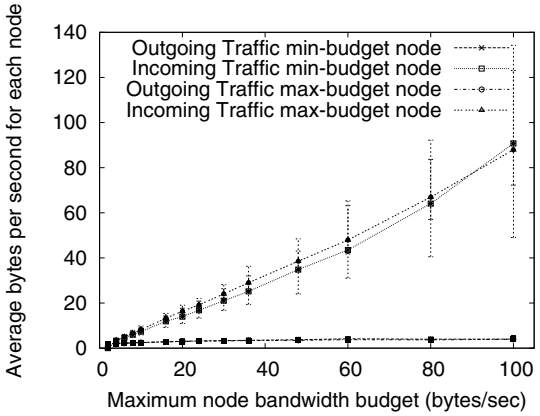


Figure 15: Bandwidth consumption of Accordion nodes in a 3000-network using a churn intensive workload where nodes have heterogeneous bandwidth budgets, as a function of the largest node's budget. For each experiment, nodes have budgets uniformly distributed between 2 and the x-value. This figure shows the consumption of the nodes with both the minimum and the maximum budgets.

higher with uniform and exponential distributions than they are with Pareto. However, Accordion still provides lower lookup latencies than Chord, except when bandwidth is very limited.

5.9 Bandwidth Control

An Accordion node does not have direct control over all of the network traffic it generates and receives, and thus does not always keep within its bandwidth budget. A node must always forward primary lookups, and must acknowledge all exploration packets and lookup requests in order to avoid appearing to be dead. This section evaluates how much Accordion exceeds its budget.

Figure 14 plots bandwidth consumed by Accordion as a function of lookup traffic rate, when all Accordion nodes have a bandwidth budget of 6 bytes/sec. The figure shows the median of the per-node averages over the life of the experiment, along with the 10^{th} and 90^{th} percentiles, for both incoming and outgoing traffic. When lookup traffic is low, nodes achieve exactly 6 bytes/sec. As the rate of lookups increases, nodes explore less often and issue fewer parallel lookups. Once the lookup rate exceeds one every 25 seconds there is too much lookup traffic to fit within the bandwidth budget. Each lookup packet and its acknowledgment cost approximately 50 bytes in our simulator, and our experiments show that at high lookup rates, lookups take nearly 3.6 hops on average (including the direct reply to the query source). Thus, for lookup rates higher than 0.04 lookups per second, we expect lookup traffic to consume more than $50 \cdot 3.6 \cdot 0.04 = 7.2$ bytes per node per second, leading to the observed increase in bandwidth.

The nodes in Figure 14 all have the same bandwidth budget. If different nodes have different bandwidth budgets, it might be the case that nodes with large budgets force low-budget nodes to exceed their budgets. Accordion addresses this issue by explicitly biasing lookup and exploration traffic towards neighbors with high budgets. Figure 15 shows the relationship between the spread of budgets and the actual incoming and outgoing bandwidth incurred by the lowest- and highest-budget nodes. The node budgets are uniformly spread over the range $[2, x]$ where x is the maximum budget shown on the x-axis of Figure 15. Figure 15 shows that the bandwidth used by the lowest-budget node grows very slowly with the maximum budget in the system; even when there is a factor of 50 difference between the highest and lowest budgets, the lowest-budget node exceeds its budget only by a factor of 2. The node with the maximum budget stays within its budget on average in all cases.

6 Related Work

Unlike other DHTs, Accordion is not based on a particular data structure and as a result it has great freedom in choosing the size and content of its routing table. The only constraint it has is that the neighbor identifiers adhere to the small-world distribution [13]. Accordion has borrowed routing table maintenance techniques, lookup techniques, and inspiration from a number of DHTs [9–11, 20, 23, 25], and shares specific goals with MSPastry, EpiChord, Bamboo, and Symphony.

Castro et al. [2] present a version of Pastry, MSPastry, that self-tunes its stabilization period to adapt to churn and

achieve low bandwidth. MSPastry also estimates the current failure rate of nodes, using historical failure observations. Accordion shares the goal of automatic tuning, but focuses on adjusting its table size as well as adapting the rate of maintenance traffic.

Instead of obtaining new state by explicitly issuing lookups for appropriate identifiers, Accordion learns information from the routing tables of its neighbors. This form of information propagation is similar to classic epidemic algorithms [6]. EpiChord [15] also relies on epidemic propagation to learn new routing entries. EpiChord uses parallel iterative lookups, as opposed to the parallel recursive lookups of Accordion, and therefore is not able to learn from its lookup traffic according to the identifier distribution of its routing table.

Bamboo [22], like Accordion, has a careful routing table maintenance strategy that is sensitive to bandwidth-limited environments. The authors advocate a fixed-period recovery algorithm, as opposed to the more traditional method of recovering from neighbor failures reactively, to cope with high churn. Accordion uses an alternate strategy of actively requesting new routing information only when bandwidth allows. Bamboo also uses a lookup algorithm that attempts to minimize the effect of timeouts, through careful timeout tuning. Accordion avoids timeouts by predicting the liveness of neighbors and using parallel lookups.

Symphony [19] is a DHT protocol that also uses a small-world distribution for populating its routing table. While Accordion automatically adjusts its table size based on a user-specified bandwidth budget and churn, the size of Symphony's routing table is a protocol parameter. Symphony acquires the desired neighbor entries by explicitly looking up identifiers according to a small-world distribution. Accordion, on the other hand, acquires new entries by learning from existing neighbors during normal lookups and active exploration. Existing evaluations of Symphony [19] do not explicitly account for bandwidth consumption nor the lookup latency penalty due to timeouts. Mercury [1] also employs a small-world distribution for choosing neighbor links, but optimizes its tables to handle scalable range queries rather than single key lookups.

A number of file-sharing peer-to-peer applications allow the user to specify a maximum bandwidth. Gia [3] exploits that information to explicitly control the bandwidth usage of nodes by using a token-passing scheme to approximate flow control.

7 Conclusion

We have presented Accordion, a DHT protocol with a unique design that automatically adjusts itself to reflect current operating environments and a user-specified bandwidth budget. By learning about new routing state opportunistically through lookups and active search, and evicting state based on liveness probability estimates, Accordion adapts its routing table size to achieve low lookup latency while staying within a user-specified bandwidth budget.

A self-tuning, bandwidth-efficient protocol such as Accordion has several benefits. Users often don't have the expertise to tune every DHT parameter correctly for a given operating environment; by providing them with a single, intuitive parameter (a bandwidth budget), Accordion shifts the burden of tuning from the user to the system. Furthermore, by remaining flexible in its choice of routing table size and content, Accordion can operate efficiently in a wide range of operating environments, making it suitable for use by developers who do not want to limit their applications to a particular network size, churn rate, or lookup workload.

Currently, we are instrumenting DHash [5] to use Accordion. Our p2psim version of Accordion is available at: `http://pdos.lcs.mit.edu/p2psim`.

Acknowledgments

We thank Frank Dabek for the King dataset measurements, Russ Cox and Thomer Gil for their help writing the simulator, and Anjali Gupta for implementing OneHop in p2psim. We are grateful to David Karger, Max Krohn, Athicha Muthitacharoen, Emil Sit, the OceanStore group at UC Berkeley, and the anonymous reviewers for their insightful comments on previous drafts of this paper. Our shepherd, Miguel Castro, provided valuable feedback that helped improve this paper.

References

[1] BHARAMBE, A. R., AGRAWAL, M., AND SESHAN, S. Mercury: Supporting scalable multi-attribute range queries. In *Proceedings of the 2004 SIGCOMM* (Aug. 2004).

[2] CASTRO, M., COSTA, M., AND ROWSTRON, A. Performance and dependability of structured peer-to-peer overlays. In *Proceedings of the 2004 DSN* (June 2004).

[3] CHAWATHE, Y., RATNASAMY, S., BRESLAU, L., LANHAM, N., AND SHENKER, S. Making Gnutella-like P2P systems scalable. In *Proceedings of the 2003 SIGCOMM* (August 2003).

[4] DABEK, F., COX, R., KAASHOEK, F., AND MORRIS, R. Vivaldi: A decentralized network coordinate system. In *Proceedings of the 2004 SIGCOMM* (Aug. 2004).

[5] DABEK, F., LI, J., SIT, E., ROBERTSON, J., KAASHOEK, M. F., AND MORRIS, R. Designing a DHT for low latency and high throughput. In *Proceedings of the 1st NSDI* (March 2004).

[6] DEMERS, A., GREENE, D., HAUSER, C., IRISH, W., LARSON, J., SHENKER, S., STURGIS, H., SWINEHART, D., AND TERRY, D. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th PODC* (Aug. 1987).

[7] GUMMADI, K. P., GUMMADI, R., GRIBBLE, S. D., RATNASAMY, S., SHENKER, S., AND STOICA, I. The impact of DHT routing geometry on resilience and proximity. In *Proceedings of the 2003 SIGCOMM* (Aug. 2003).

[8] GUMMADI, K. P., SAROIU, S., AND GRIBBLE, S. D. King: Estimating latency between arbitrary Internet end hosts. In *Proceedings of the 2002 SIGCOMM Internet Measurement Workshop* (Nov. 2002).

[9] GUPTA, A., LISKOV, B., AND RODRIGUES, R. Efficient routing for peer-to-peer overlays. In *Proceedings of the 1st NSDI* (Mar. 2004).

[10] GUPTA, I., BIRMAN, K., LINGA, P., DEMERS, A., AND VAN RENESSE, R. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proceedings of the 2nd IPTPS* (Feb. 2003).

[11] KAASHOEK, M. F., AND KARGER, D. R. Koorde: A simple degree-optimal hash table. In *Proceedings of the 2nd IPTPS* (Feb. 2003).

[12] KARGER, D., LEHMAN, E., LEIGHTON, F., LEVINE, M., LEWIN, D., AND PANIGRAHY, R. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th STOC* (May 1997).

[13] KLEINBERG, J. The small-world phenomenon: An algorithmic perspective. In *Proceedings of the 32nd STOC* (May 2000).

[14] KRISHNAMURTHY, S., EL-ANSARY, S., AURELL, E., AND HARIDI, S. A statistical theory of chord under churn. In *Proceedings of the 4th IPTPS* (Feb. 2005).

[15] LEONG, B., LISKOV, B., AND DEMAINE, E. D. Epichord: Parallelizing the Chord lookup algorithm with reactive routing state management. In *Proceedings of the 12th International Conference on Networks* (Nov. 2004).

[16] LI, J., STRIBLING, J., MORRIS, R., KAASHOEK, M. F., AND GIL, T. M. A performance vs. cost framework for evaluating DHT design tradeoffs under churn. In *Proceedings of the 24th INFOCOM* (Mar. 2005).

[17] LIBEN-NOWELL, D., BALAKRISHNAN, H., AND KARGER, D. R. Analysis of the evolution of peer-to-peer systems. In *Proceedings of the 21st PODC* (Aug. 2002).

[18] LITWIN, W., NEIMAT, M.-A., AND SCHNEIDER, D. A. LH* — a scalable, distributed data structure. *ACM Transactions on Database Systems 21*, 4 (Dec. 1996), 480–525.

[19] MANKU, G. S., BAWA, M., AND RAGHAVAN, P. Symphony: Distributed hashing in a small world. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)* (Mar. 2003).

[20] MAYMOUNKOV, P., AND MAZIÈRES, D. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of the 1st IPTPS* (Mar. 2002).

[21] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content addressable network. In *Proceedings of the 2001 SIGCOMM* (Aug. 2001).

[22] RHEA, S., GEELS, D., ROSCOE, T., AND KUBIATOWICZ, J. Handling churn in a DHT. In *Proceedings of the 2004 USENIX Technical Conference* (June 2004).

[23] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)* (Nov. 2001).

[24] SAROIU, S., GUMMADI, P. K., AND GRIBBLE, S. D. Measuring and analyzing the characteristics of Napster and Gnutella hosts. *Multimedia Systems Journal 9*, 2 (Aug. 2003), 170–184.

[25] STOICA, I., MORRIS, R., LIBEN-NOWELL, D., KARGER, D. R., KAASHOEK, M. F., DABEK, F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking 11*, 1 (Feb. 2003), 149–160.

[26] ZHAO, B. Y., HUANG, L., STRIBLING, J., RHEA, S. C., JOSEPH, A. D., AND KUBIATOWICZ, J. D. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications 22*, 1 (Jan. 2004), 41–53.

Improving Web Availability for Clients with MONET

David G. Andersen
Carnegie Mellon University
dga@cs.cmu.edu

Hari Balakrishnan, M. Frans Kaashoek, Rohit N. Rao
MIT Computer Science and Artificial Intelligence Lab
{hari,kaashoek,rohit}@csail.mit.edu

http://nms.csail.mit.edu/ron/ronweb

Abstract

Despite the increasing degree of multi-homing, path and data redundancy, and capacity available in the Internet, today's clients experience outage rates of a few percent when accessing Web sites. MONET ("Multi-homed Overlay NETwork)*, is a new system that improves client availability to Web sites using a combination of link multi-homing and a cooperative overlay network of peer proxies to obtain a diverse collection of paths between clients and Web sites. This approach creates many potential paths between clients and Web sites, requiring a scalable way to selecting a good path. MONET solves this problem using a* waypoint selection *algorithm, which picks a good small subset of all available paths to actively probe.*

MONET runs on FreeBSD, Linux, and Mac OS X, and is deployed at six different sites. These installations have been running MONET for over one year, serving about fifty users on a daily basis. Our analysis of proxy traces shows that the proxy network avoids between 60% and 94% of observed failures, including access link failures, Internet routing problems, persistent path congestion, and DNS failures. The proxy avoids nearly 100% of failures due to client and wide-area network failures, with negligible overhead.

1 Introduction

Web clients experience failure rates as high as a few percent when attempting to connect to Web sites today. To improve this situation, many techniques have been proposed: client-side *multi-homing*, in which the client's access to the Internet uses multiple links, deploying and using redundant paths in the Internet, server-side multi-homing, and server replication. These methods do help, but previous work [10, 7] and our results (Section 4) demonstrate that the resulting availability, defined as the fraction of time that a service is reachable and working, is between 95% and a little over 99%. To put these numbers in perspective, consider the availability figures for the U.S. public telephone system (over 99.99% [26, 19, 13]) and the emergency telephone service (99.994% in 1993 [25]).

The "1.5-2 nines" of availability of current Internet-based systems makes them unattractive for important applications such as medical collaborations and certain financial transactions, both of which often use expensive, dedicated networks today in order to provide the required availability. The desire for high availability is not limited to so-called critical applications—any downtime is expensive for businesses that conduct transactions over the Internet [41]. Even brief interruptions lasting more than a few seconds can degrade user perception of a site's performance and lead to substantial revenue losses.

We seek to improve the availability of client accesses to Web sites by an order of magnitude (one more "nine") or better. We restrict our attention to the Web to make the problem focused and tractable. Despite this narrowed focus, the problem remains challenging, because there are many components whose failure can prevent a client from reaching a Web site. The client's access link may be down; the Domain Name System (DNS) may not respond or may have incorrect information [17]; misconfigurations [22], congestion, and routing pathologies [29] might make the network path between client and server unavailable; or the server itself or its access network may be down. Many of these failures are unpredictable, silent, and have complex root causes.

We propose *MONET (Multi-homed Overlay Network)*, a system that improves Web site availability for clients. Web clients use MONET as a standard Web proxy. MONET attempts to mask failures by obtaining and exploring multiple different end-to-end paths for each HyperText Transfer Protocol (HTTP) request. To help mask failures at different locations in the Internet, MONET finds these paths in three ways: (a) link multi-homing; (b) forwarding requests and responses via a small overlay network of peer MONET proxies; and (c) contacting multiple server replicas. MONET explores paths using probes that check the availability of multiple underlying components.

MONET's end-to-end approach recovers from a variety of failures of the individual components involved in an HTTP request. MONET's protocols and algorithms detect and respond to failures within a small number of round-trips, and with low overhead, sending only a few additional packets. It detects failures regardless of their root cause, providing a measure of resilience against not only network-layer faults, but also persistent congestion, active attacks, misconfiguration, DNS outages, and server-side failures.

MONET uses a *waypoint selection algorithm* that dynam-

ically decides the order in which the many possible paths between client and server should be used, and at what time to use any given path. The algorithm determines this ordering by maintaining statistics about path success rates and connection times through different interfaces and peers. By pruning the large space of possible paths to a handful of the most promising ones, this algorithm reduces MONET's overhead on the network and on Web sites to tolerable levels.

This paper describes a version of MONET that has been in daily use by over fifty people (a conservative estimate; the MONET logs anonymize user activity) at MIT CSAIL since Sept. 2003. The CSAIL proxy is multi-homed to three different ISPs and uses five other peer proxies at different Internet locations.

Our analysis of trace data collected from the MONET installations shows that MONET overcomes at least 60% of all outages (Table 3) and nearly *all* non-server failures (Figure 10), while imposing little overhead . We found that access link failures, wide-area failures, and server-side failures all contributed to the lack of availability and had to be masked. While multi-homing a service alone does not increase its availability (Figure 11), using MONET *in conjunction* with server multi-homing greatly increases availability. This increase arises because MONET reduces client and wide-area failures, *and* because MONET actively seeks out multiple paths to multi-homed sites. MONET achieves significant ("one to two nines") availability improvements at modest cost; for instance, MONET can use a cheap DSL line to greatly increase the availability of a site that already uses BGP multi-homing.

These benefits are tempered by some limitations of the current system. If the different paths available between a proxy and server all share a single point of failure (*e.g.*, a particular network link, a misconfigured DNS database, etc.), MONET will not mask the failure of that element. The current MONET implementation does not mask mid-stream failures that might occur in the middle of a TCP connection; such failures may be recovered from by issuing appropriate HTTP range requests or using transport-layer techniques.

2 MONET Design

MONET consists of a set of Web proxies deployed across the Internet, which serve as conduits for client connections to Web sites. One site might have one or a few proxies, and the entire system a handful to a few hundred proxies.

The goal of MONET is to reduce periods of downtime and exceptional delays that lead to a poor user experience. The idea is to take advantage of several redundant client to server paths, whose failure modes are expected to be mostly independent. MONET must therefore address two questions:

1. How to obtain multiple paths from a client to a site?
2. Which path(s) to use, and at what times?

The answers are shaped by three requirements:

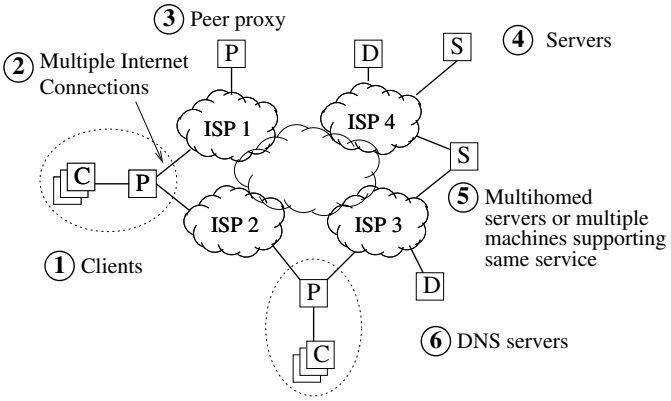


Figure 1. The MONET environment. Clients (1) contact Web sites via a local MONET proxy. That local proxy may be multi-homed with multiple local interfaces (2), and may also route requests through remote peer proxies (3). Clients wish to communicate with web sites (4), which may be themselves multi-homed or spread over multiple machines (5). Web sites must be located using DNS (6); DNS servers are typically replicated over multiple machines.

- **R1** The network overhead introduced by MONET in terms of the number of extra packets and bytes must be low.
- **R2** The overhead imposed on Web servers in terms of TCP connections and data download requests must be low.
- **R3** When possible, MONET should improve user-perceived latency, by reducing the tail of the latency distribution and balancing load on multi-homed links.

The first two requirements preclude an approach that simply attempts concurrent connection requests along all paths between a proxy and Web site.

2.1 Obtaining Multiple Paths

Each proxy has some of the following paths to a Web site at its disposal, as shown in Figure 1. The term *path* refers either to a direct Internet path from one IP address to another, or to an indirect path that goes through an intermediate node.

2.1.1 Multi-homed Local Interfaces

A MONET proxy can obtain Internet access via multiple Internet Service Providers (ISPs), ideally at least two, and perhaps three or four. The proxy can then use a subset of these *local interfaces*, either concurrently or serially, to resolve DNS names and to connect to Web sites. The MONET proxy is assigned one IP address from each upstream ISP, allowing it to direct requests through any chosen provider. This "host-based" multi-homing approach works particularly well for MONET proxies in smaller organizations, providing them the benefits of multi-homing without the complexity of BGP configuration and management.

MONET initiates several TCP connections (sending TCP SYNs) to the server both to probe and to establish a connection over which to request data. The proxy then directs requests only along a link over which a connection succeeded.

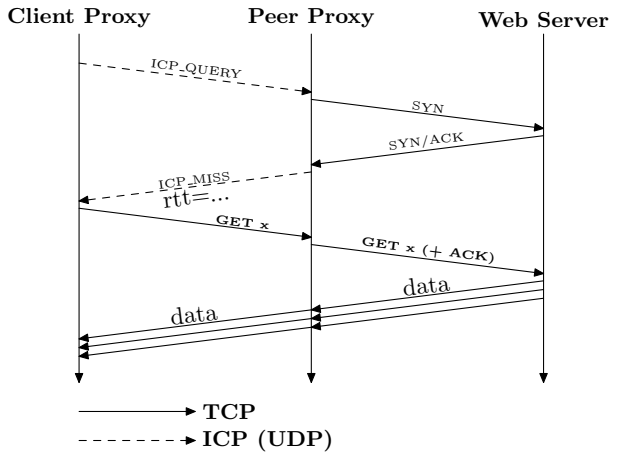


Figure 2. The ICP+ protocol probing a path through a peer. The peer proxy uses a cached DNS response for the site. After the ICP+ probe, the client proxy sends via TCP a request to the peer proxy to fetch the object; the peer retrieves this data over the TCP connection that it used to probe the site during the first exchange.

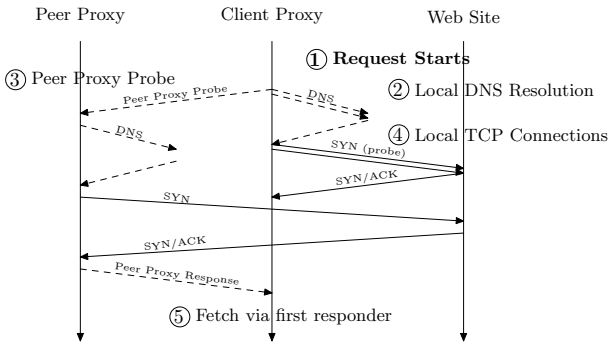


Figure 3. The client proxy performs several queries in parallel. When the request begins (1), it simultaneously begins DNS resolution (2), and contacts peer proxies for the object (3). After DNS resolution has completed, the MONET proxy attempts TCP connections (delayed by the output of the waypoint selection step) to the remote server via multiple local interfaces (4). The remote proxy performs the same operations and returns a reply to the client proxy. The MONET proxy retrieves the data via the local or indirect path that responded first.

This dual use of the TCP SYN packets reduces network overhead, and is an effective tactic for choosing between a set of replicated servers [11]. Only one of these connections will be used to retrieve data.

2.1.2 Paths Through Peer Proxies

An overlay network is a convenient way of obtaining access to multiple paths between two end points, allowing many Internet path failures to be masked [7]. Building upon this observation, MONET attempts to find additional paths using an overlay network of peer proxies. To let MONET probe the availability of these paths, we designed ICP+, a backward-compatible extension to the Inter-Cache Protocol (ICP) [38].

ICP checks whether an object is in a peer's cache. ICP+ extends this check by optionally asking the peer to probe the origin server using a TCP SYN, as described earlier, and return the round-trip connection establishment time. The client proxy can then request the object via a TCP connection to the peer proxy. Figure 2 depicts the operation of ICP+.

An ICP+ query includes the URL of the object that the proxy wants to retrieve through a peer proxy. A peer proxy handles ICP+ queries just like requests from its clients, but the proxy does not contact other peer proxies in turn. MONET proxies handle ICP+ queries as follows:

1. If the object is cached, then reply immediately.
2. If an open connection to the server exists, then reply with that connection's establishment RTT.
3. Else, resolve DNS, perform waypoint selection, ignore other peer paths;
   Open a connection to the server;
   Reply with RTT when TCP established.

The operation of a proxy with one peer proxy is illustrated in Figure 3. This diagram shows one additional benefit of performing the ICP+ queries in parallel with sending TCP SYNs to the origin server: it eliminates delays that the proxy would ordinarily experience waiting for ICP replies. If the ICP replies for a cached object are delayed, the client proxy might fetch the object directly, which is the correct behavior if the origin server is closer than the peer proxy.

2.1.3 Multi-homed Web Sites

Web sites are sometimes replicated on distinct hosts, or are multi-homed using different links to the Internet. The DNS name for a replicated site is often bound to multiple IP addresses. MONET considers each address as corresponding to a different server machine or Internet path, although portions of the paths may be shared (we believe that configurations that deliberately violate this assumption are rare).

Today's Web clients typically contact only one address for a Web site, or they wait between 30 seconds and 13 minutes before contacting subsequent addresses. Because they cannot count on clients to quickly fail over, Web site administrators rely on one of two mechanisms to direct clients to a working server. Many sites use front-end load distributors to direct clients to a host in a cluster. Others answer DNS queries with responses that have very low TTL (time to live) values, forcing clients to frequently refresh the name-to-address mapping for the site. If a server fails, the DNS server stops announcing the failed address. MONET masks failures on shorter timescales without requiring Web sites to set low TTLs in their DNS records.

2.1.4 Multi-path DNS Resolution

A MONET proxy performs at least two concurrent DNS requests (on different local interfaces) to mask DNS failures for two reasons. First, DNS servers are—or should be—

replicated, so finding multiple paths is easy. Second, sending multiple DNS requests does not cause high network overhead because DNS lookups are much less frequent than TCP connections: in our Web traces, 86% of the connections from the deployed MONET proxy to remote servers used a cached DNS entry. This number is consistent with other studies of DNS and TCP workloads [17], which estimated that overall cache hit rates were between 70 and 80%.

Because some server-side content distribution services return DNS responses tailored to a client's location in the network, a MONET proxy performs DNS resolution using only its local interfaces. Each peer proxy performs its own DNS resolution. This localized resolution helps the MONET proxy fetch data from a replica near it.

2.2 Choosing Paths: Waypoint Selection

If a MONET proxy has ℓ local links and r single-homed peer proxies it can use, and if the site has s IP addresses, then the total number of potential paths to the Web site at the proxy's disposal is $\ell \cdot s$ direct paths plus $\ell \cdot r \cdot s$ indirect paths. If each peer proxy has ℓ local interfaces of its own, then the number of paths increases to $\ell \cdot s$ direct paths plus $\ell^3 \cdot r \cdot s$ indirect paths. For even moderate values of ℓ, r, and s, this number is considerable; *e.g.*, when $\ell = 3$, $r = 10$, and $s = 2$, the number of possible paths is 546. When the peer proxies are single-homed, this number is 66, still quite large.

Of course, not all of these paths are truly independent of each other, and pairs of paths may actually share significant common portions. Each path, however, has something different from all the other paths in the set. MONET uses waypoint selection to pick subsets of its paths to probe at different times.

The waypoint selection algorithm takes the available local interfaces, peer-proxy paths, and target Web site IP addresses, and produces an ordered list of these interfaces and paths. Each element of this list is preceded by an optional delay that specifies the time that should elapse before the corresponding path is probed. The proxy attempts to connect to the server(s) in the specified order. The waypoint selection algorithm seeks to order paths according to their likelihood of success, but it must also occasionally attempt to use paths that are not the best to determine whether their quality has changed. MONET attempts these secondary paths in parallel with the first path returned by waypoint selection. If the measured path connects first, MONET uses it as it would any other connection.

Waypoint selection is superficially similar to classical server selection in which a client attempts to pick the best server according to some metric. Under waypoint selection, however, a client can use its history of connections to a variety of servers along different paths to infer whether or not those paths are likely to be functioning, and what the path loss probabilities are. Then, when confronted with a request involving a new server, the client can decide which of its paths are best suited to retrieve data.

2.2.1 Which Paths to Probe

MONET ranks its local links and local link-remote proxy pairs using an exponential weighted moving average (EWMA) of the success rate (fraction of probes that received a response within a timeout period) along each of these paths. It breaks ties using average response time. The algorithm updates the success rate for a local link a short time after sending a TCP SYN or DNS request using that link. Similarly, ICP+ queries update the statistics for the particular local link-proxy pair through which the query was sent.

The proxy sends all DNS requests both on the local link with the highest success rate and also via a randomly selected second local link. The proxy also attempts an additional TCP SYN to the site or sends an ICP+ query to a random peer via a random link between 1% and 10% of the time to measure infrequently used paths.

In designing MONET's waypoint selection algorithm, we considered only schemes that rank the local links and peer proxy paths, regardless of which servers were previously accessed along the various paths. Grouping the success rates by remote site name or IP prefix might yield additional benefit.

2.2.2 When to Probe Paths

To keep overhead small, a MONET proxy should perform the next request attempt only when it is likely that each prior attempt has failed. The delay between requests on different paths must be long enough to ensure this behavior, but short enough so that requests are fulfilled without undue delay. This delay should adapt to changing network conditions.

Measurements of round-trip connect times from the operational MONET proxy at MIT show that their distribution is multi-peaked (the "knee" on the CDF, and the peaks in the histogram in Figure 4), suggesting that the best delay threshold is just after one of the peaks. For example, in this figure, very few arrivals occur between 0.6 and 3.1 seconds; increasing the threshold past 0.6 seconds increases delay without significantly reducing the chances of a spurious probe.

We explored two ways of estimating this delay threshold:

1. **k-means clustering.** This method identifies the peaks in the connect time PDF by clustering connect time samples into k clusters, and finding a percentile cutoff just outside one of the peaks (clusters). The centroids found by k-means with $k = 16$ are shown as horizontal lines in Figure 4. The clustering is relatively insensitive to the value of k.

This method is computationally expensive, particularly if the clustering is recomputed each time a connection attempt succeeds or fails. Even when the threshold is only recomputed periodically, the computational load and memory requirements may exceed what is acceptable for a busy proxy: the k-means clustering requires that the proxy maintain a large history of previous probes.

2. ***rttvar*-based scheme.** To avoid the cost of the k-means scheme, we considered an *rttvar* scheme inspired by TCP retransmission timers. Each delay sample, independent of

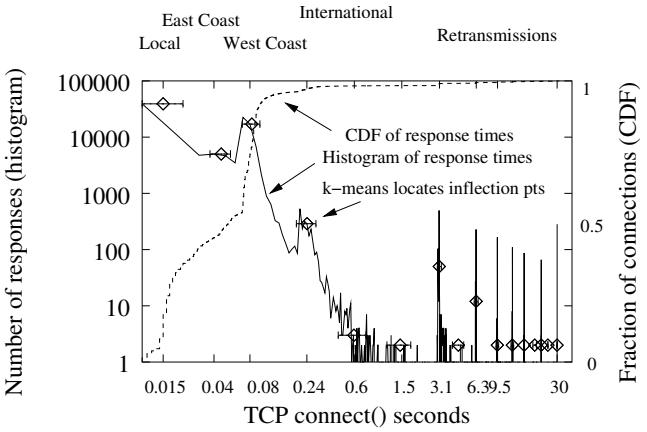


Figure 4. k-means clustering applied to TCP connect times for 137,000 connections from one access link on the east coast to sites around the world. The CDF shows the cumulative fraction of requests amassed by the histogram.

the server contacted or path used, updates an EWMA estimate of the average delay (*rtt*) and another EWMA estimate of the average linear deviation of the delay (*rttvar*). The delay threshold between subsequent requests is set to $rtt + 4 \cdot rttvar$.

The *rttvar* scheme is substantially simpler to calculate than k-means clustering, but it may pick a threshold in the middle of a "valley" between two peaks in the delay sample distribution. In practice, measurements from MONET (*e.g.*, the data illustrated in Figure 4) show that *rttvar* estimates an 800 ms delay threshold, while k-means estimates thresholds of 295 ms (2% false transmission probability), 750 ms (1.6%), and 3.2s (1%). A MONET using the 2% k-means estimator would decide that its first connection had failed after 300 ms instead of 800 ms, reducing the fail-over time for the failed connection. We do not believe that this modest latency improvement justifies the complexity and increased computational and storage requirements of the k-means estimation, and so we chose the rttvar scheme for MONET.

2.3 The Client-MONET Interface

Clients specify a set of MONET nodes, preferably nodes that are close to them in the network, as their Web proxies (one proxy is the primary and the rest are backups). The proxy-based approach allows MONET to be easily and incrementally deployed within an organization, and has been essential to attracting users and gathering data using live user traffic. In addition to ease of deployment, we chose the proxy approach because it provides two other significant benefits:

1. Path information: Because a MONET proxy observes what *site* clients want to contact (such as `www.example.com`), instead of merely seeing a destination IP address, it has access to several more paths for the waypoint selection algorithm to consider when the site is replicated across multiple IP addresses. Moreover, by operating at the application layer and resolving the DNS name of a site to its IP addresses, MONET is able to mask DNS errors; such errors are a non-negligible source of client-perceived site outages and long delays [17, 9].

2. Access control: Many sites control access to content based upon the originating IP address, which is changed by using a different local link or by transiting through a remote proxy. Early users of MONET were occasionally unable to access material in licensed scientific journals, because the proxy had redirected their access through a non-licensed IP address. The deployed MONET proxy is now configured to direct access to 180 licensed web sites through a local interface. As with the CoDeeN proxies [27], this approach also ensures that clients cannot gain unauthorized access to licensed content via MONET.

2.4 Putting it All Together

When presented with a client's request for a URL, MONET follows the procedure shown in Figure 5. The MONET proxy first determines whether the requested object is cached locally. If not, then the proxy checks to see whether the site has successfully been contacted recently, and if so, uses an open TCP connection to it, if one already exists.[1]

Otherwise, the proxy uses MONET's *waypoint selection* algorithm to obtain an ordered list of the available paths to the site. This list is in priority order, with each element optionally preceded by a delay. The proxy attempts to retrieve the data in the order suggested by this list, probing each path after the suggested delay.

If waypoint selection lists a peer proxy first, the request is issued immediately. MONET concurrently resolves the site's DNS name to its corresponding IP addresses to determine which paths are available for local interfaces. To mask DNS failures, the proxy attempts this resolution using all of its local interfaces.

After resolving the domain name, the proxy sends TCP SYN probes from the selected local interfaces. The proxy retrieves data from the first probe (SYN or peer-proxy request) that responds. The results of the DNS lookups and path probes update information about path quality maintained by the waypoint selection algorithm.

The MONET approach to masking failures operates on three different time-scales to balance the need to adapt rapidly with the desire for low overhead. The slowest adaptation (days or weeks) involves the deployment of multi-homed local links and peer proxies in different routing domains. Currently, this configuration is updated manually; automating it is an important future task.

The intermediate time scale adaptation, waypoint selection, maintains a history of success rates on the different paths, allowing MONET to adapt the order of path exploration on a time-scale of several seconds.

To respond to failures within a few round-trip times, the proxy generally attempts the first two paths returned by waypoint selection within a few hundred milliseconds, probing

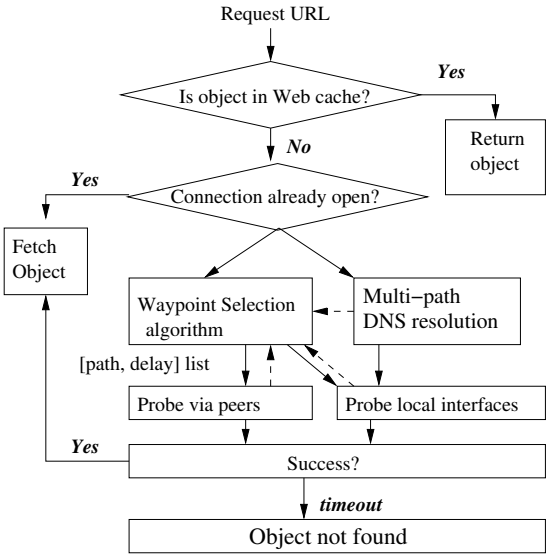


Figure 5. Request processing in MONET.

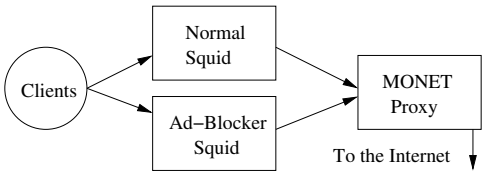


Figure 6. The squid configuration

the rest of the paths within a few seconds. If this order is good, the chances of a successful download via one of the probed paths is high, since the probe includes setting up the connection to the destination site.

Once the proxy has established the connection for a request, it uses the same path. MONET could mask mid-stream failures during large transfers by, for example, issuing an HTTP range request to fetch the remaining content, but the current implementation does not do so. Typical Web workloads consist of many smaller objects, so mid-stream failover will not make much difference for most connections.

3 Implementation

The MONET proxy is implemented as a set of changes to the Squid Web proxy [33] and the `pdnsd` parallel DNS resolver [24], along with a set of host policy routing configurations to support explicit multi-homing. MONET runs under FreeBSD, Linux, and Mac OS X, and should run any POSIX-compliant system that provides a way to support explicit multi-homing.

In the deployed system, Web client configurations are specified with Javascript that arranges for a suitable backup proxy from the specified set to be used if the primary proxy fails. As an extra incentive for users to use the MONET proxy, one front-end blocks common banner ads and pop-up advertisements. Figure 6 shows the Squid configuration.

Because we wanted to evaluate multiple waypoint selection algorithms, the deployed proxy probes *all* of its paths in parallel without performing waypoint selection. We then used subsets of this all-paths data to determine the performance of the waypoint selection algorithms. The currently deployed waypoint selection algorithm returns a static list of (path, delay) pairs that it chooses based upon the name of the destination Web site, to address the access control problems mentioned in Section 2.3.

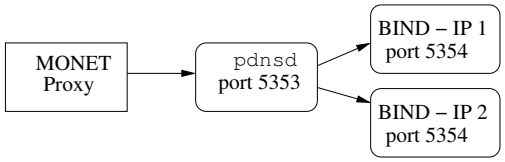


Figure 7. The DNS configuration. `pdnsd` sends queries in parallel to each BIND server, which resolves the query independently.

3.1 Explicit Multi-homing

The MONET proxy and DNS server explicitly bind to the IP address of each physical interface on the machine. MONET uses FreeBSD's `ipfw` firewall rules or Linux's policy routing to direct packets originating from a particular address through the correct upstream link for that interface.

The MONET proxy communicates with a front-end DNS server, `pdnsd`, running on a non-standard high port. `pdnsd` is a DNS server that does not recursively resolve requests on its own, but instead forwards client requests to several recursive DNS servers in parallel—in our case, to BIND, the Berkeley Internet Name Daemon [5]. An instance of BIND runs on each local interface, as shown in Figure 7. This configuration resolves each DNS query independently on each of the outbound interfaces, so that we can confirm during analysis whether the query would have succeeded or failed if sent on that interface alone. Each BIND resolves the query independently, and rotates through the list of available name servers. Because most domains have at least two name servers [12], MONET usually copes with the failure of one of its links or of a remote DNS server without delay.

3.2 ICP+ with Connection Setup

ICP+ adds two new flags to the `ICP_QUERY` message: `ICP_FLAG_SETUP` and `ICP_FLAG_SETUP_PCONN`. A query with `ICP_FLAG_SETUP` requests that the remote proxy attempt a TCP connection to the origin server before returning an `ICP_MISS`. Peer caches that do not support ICP+—or do not wish to provide ICP+ to that client—simply ignore the flag and reply with standard ICP semantics. Squid supports a mechanism for occasionally sending ICMP ping packets to origin servers, using ICP's option data field to return that ping time in response to an ICP query. ICP+ piggybacks upon this mechanism to return the measured RTT from connection initiation.

Because it is used for probing network conditions, ICP+ uses unreliable UDP datagrams to communicate between

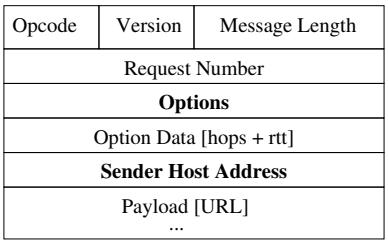


Figure 8. The ICP Packet Format. Bold indicates the fields extended to support ICP+. Brackets show the contents of the fields for Web proxy communication.

peer proxies. Using UDP avoids mistaking temporary failures and packet loss for increased latency, as would happen if a reliable transport protocol like TCP were used for the probes. To treat local interfaces and peer-proxy paths consistently, MONET retransmits lost ICP+ messages with the same 3-second timer that TCP uses for its initial SYN packets. Once a peer has confirmed access to a Web site, the proxies use TCP to transmit objects between them.

MONET uses Squid's persistent connection cache to reduce connection setup overhead. If the originating proxy has a persistent connection open to a Web site, it bypasses peer selection and directly uses the persistent connection, on the assumption that in one of the previous selection attempts, its own connection seemed best. When a remote proxy has a persistent connection to the origin server, it responds immediately to ICP queries, setting the `ICP_FLAG_SETUP_PCONN` flag, and supplying the RTT from when it initially opened the connection.

Figure 8 shows the ICP packet header with the MONET additions in bold. RFC 2187 notes that the sender host address is normally zero-filled. ICP+ uses this field and the request number to suppress duplicates. A multi-homed MONET proxy can transmit multiple ICP+ probes to a peer, from each of its local interfaces to each of the peer's interfaces. On startup, each MONET proxy picks a 32-bit number as its sender ID (*e.g.*, a random number or a local interface address), and uses the same ID when sending via any of its interfaces. The $(sender\ ID, request\ \#)$ tuple uniquely identifies each request and allows a peer proxy to not send multiple identical requests to a Web server. This mechanism provides additional redundancy between proxies without imposing additional server overhead.

Finally, we note that ICP's lack of authentication causes several known security flaws. The newer UDP-based HyperText Caching Protocol (HTCP) [37] supports strong authentication of requests. HTCP requests also carry request attributes such as cookies that may affect whether an object can be served from cache or not. Our HTCP-based MONET is functionally identical to the ICP-based version. The deployed system uses the more mature ICP+ implementation.

3.3 Reducing Server Overhead

Waypoint selection greatly reduces the number of wasteful connection attempts. MONET must also ensure that the few remaining connection attempts do not unnecessarily create server state. Because modern servers minimize processing of SYN packets (to thwart denial-of-service attacks) using techniques like SYN Cookies [8] and SYN caches [21], MONET can send multiple SYN packets without incurring serious overhead, as long as *exactly one TCP three-way handshake completes*, since a connection consumes significant server resources once the server receives the final ACK in the three-way TCP handshake. After opening one connection successfully, MONET closes the remaining probe connections. If this close occurs before the kernel sent an ACK for the connection, the overhead is avoided. We have proposed a simple kernel modification that reduce the overhead even further, and enables applications to change servers at earlier points in the connection attempt [6]; we omit a detailed discussion because of space constraints.

4 Evaluation

Our experimental evaluation focuses on the number of "nines" of availability achieved with and without MONET. The number of nines does not capture all aspects of availability (such as the rate at which failures occurred and how long they lasted), but it does give a good idea of overall availability (and downtime) with and without MONET.

We address the following questions:

1. To what extent do subsystems such as DNS, access links, *etc.* contribute to failures incurred while attempting to access Web sites?
2. How well does MONET mask failures, what is its overhead, and how does it compare against an idealized (but high-overhead) scheme that explores all available paths concurrently?
3. What aspects (physical multi-homing, peer proxies, *etc.*) of MONET's design contribute to MONET's observed improvement in availability? Is MONET useful if BGP multi-homing is already used at the client?
4. How much more of an availability improvement does MONET provide if the Web site is replicated?

4.1 MONET Testbed and Data Collection

We deployed the MONET proxy at six sites from the RON testbed, listed in Table 1. This analysis examines requests sourced from two of these proxies, CSAIL and Mazu , both of which are physically multi-homed. The CSAIL proxy has three peers and uses three local links:

1. **MIT:** A 100 Mbits/s link to MIT's network. MIT's network is itself BGP multi-homed to three different upstream ISPs.
2. **Cog:** A 100 Mbits/s link from Cogent.
3. **DSL:** A 1.5 Mbits/s (downstream), 384 Kbits/s (upstream) DSL link from Speakeasy.

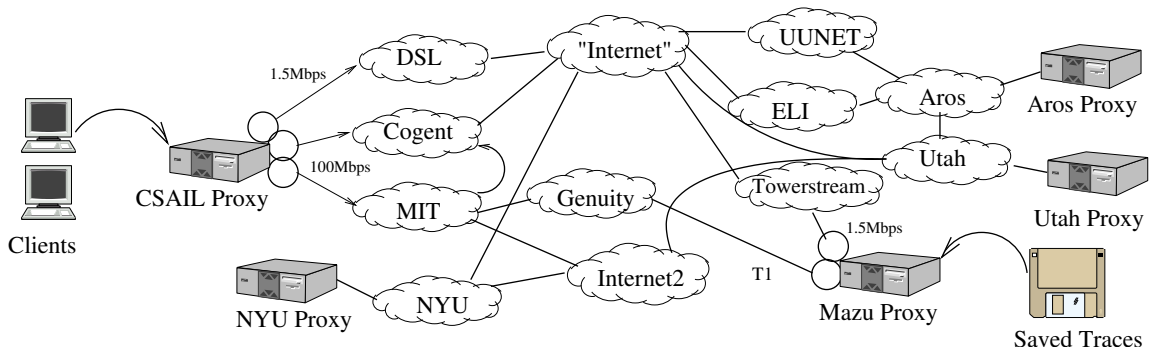


Figure 9. A partial AS-level view of the network connections between five of the deployed MONET proxies (the mediaone and CMU proxies are not shown). The CSAIL proxy peers with NYU, Utah, and Aros; the Mazu proxy peers with CSAIL, Aros, and NYU. The other sites are not directly multi-homed and do not have a significant number of local users; their traces are omitted from the analysis.

Site	Connectivity	Times
CSAIL	3: 2x100Mb, 1.5Mb DSL	6 Dec - 27 Jan 2004
Mazu	2: T1, 1.5Mb wireless	24 Jan - 4 Feb 2004
Utah	1 (US university - West)	proxy-only
Aros	1 (US local ISP - West)	proxy only
NYU	1 (US university - East)	proxy-only
Cable	2: DSL, Cable	22 Sep - 14 Oct 2004

Table 1. The sites at which the MONET proxy was deployed. Mazu's wireless connection uses a commercial wireless provider. The cable modem site was operational for one year, but was monitored only briefly.

Request type	Count
Client objects fetched	2.1M
Cache misses	1.3M
Client bytes fetched	28.5 GBytes
Cache bytes missed	27.5 GBytes
TCP Connections	616,536
Web Sessions	137,341
DNS lookups	82,957

Table 2. CSAIL proxy traffic statistics.

The Mazu proxy uses two different physical access links: a 1.5 Mbits/s T1 link from Genuity, and a 1.5 Mbits/s wireless link from Towerstream. Figure 9 shows the Autonomous Systems (AS) that interconnect our deployed proxies.

The CSAIL proxy has the largest client base, serving about fifty different IP addresses every day. It has been running since April 2003; this evaluation focuses on data collected during a six-week period from December 6, 2003 until January 27, 2004. Analysis of a second one-month period from Sep-Oct 2004 showed results similar to those presented here. Table 2 shows the traffic statistics for the CSAIL proxy.

The MONET proxies record the following events:

1. **Request time:** The time at which the client (or peer) request arrived at the proxy, and, if the request was served, the time at which the HTTP response was sent to the requester. For uncached objects, the proxy also maintains records of the following three events.
2. **DNS resolution duration:** The time at which the proxy made a request to `pdnsd`. For uncached DNS responses, the time at which DNS requests were sent on each local link, and the times at which the corresponding responses were received (if at all).
3. **TCP connect duration:** The time at which TCP SYN packets were sent on each local link and the times at which either the TCP `connect()` call completed, or a TCP connection reset (RST) packet was received.
4. **ICP+ duration:** The time at which the proxy sent an ICP+ message to a peer proxy, the time at which it was received by the peer proxy, and the time at which the ICP+ response returned.

In our experiments, when the proxy receives a request for an object from a Web site, it attempts to contact the Web site using *all* of its local interfaces and all of its peer proxies. The proxy records the time at which the original request was received and the times at which the connection establishment steps occurred using each of the local interfaces and peer proxies. Because the proxy uses all of its interfaces concurrently, the later analysis can examine the performance of a proxy that used only a subset of the interfaces. The analysis then simulates the effects of different waypoint selection algorithms by introducing various delays before additional interfaces are used.

We make a few observations about the data collected from the MONET proxies:

Caching effects: 37% of valid objects were served from the cache, saving about 3.5% of the requested bytes. As in previous studies, a few large transfers dominated the proxy's byte-count, while the majority of the sessions consisted of smaller requests. These cache hits reduce user-perceived delays, but do not mask many outages: numerous pages either required server re-validation, or included uncached objects.

Sessions: We primarily examine the success or failure of a *session*, defined as the first request to a particular server

for a Web site after 60 seconds or more of inactivity.[2]). Analyzing failures in terms of sessions rather than connections avoids a significant bias—an unreachable server generates only a *single* failed request, but a successful connection generates a stream of subsequent requests, which would give a false sense of higher availability. The proxy also uses persistent connections to fetch multiple objects from the same Web server, which reduces the total number of TCP connections. The proxy attempted 616,437 connections to external Web sites over 137,341 sessions.

Excluded objects: The following requests were excluded from analysis: Web sites within MIT, cached objects, accesses to unqualified hostnames or non-existent domain names (NXDOMAIN), access to subscription-based Web sites for which the proxy performs non-standard handling, and accesses to ten Web sites that consistently exhibited anomalous DNS or other behavior.[3] Excluding NXDOMAIN requests ignores some classes of misconfiguration-based DNS failures. Because internal network failures at the proxy site prevent users' requests from reaching the proxy, the analysis missed network failures that coincided with client failures (*e.g.*, power failures).

We do not claim that the performance of these five Internet links at MIT and Mazu represents that of a "typical" Internet-connected site. In fact, MONET would likely be used in much worse situations than those we studied to group a set of affordable low-quality links into a highly reliable system. These measurements do, however, represent an interesting range of link reliability, quality, and bandwidth, and suggest that MONET would likely benefit many common network configurations.

4.2 Characterizing Failures

The failures observed by MONET fall into five categories, listed below. We were able to precisely determine the category for each of the 5,201 failures listed in Table 3 because the links connecting the CSAIL proxy (from which the bulk of our traces are gathered) never all failed at the same time. The categories of observed failures are:

1. **DNS**: The DNS servers for the domain were unreachable or down. The originating proxy contacted multiple peer proxies, and no local links or peers could resolve the domain.
2. **Site RST**: The site was reachable because a proxy saw at least one TCP RST from a server for the site being contacted, but no connection succeeded on any local interface, and no peer proxy was able to retrieve the data. TCP RST packets indicate that the server was unable to accept the TCP connection.
3. **Site unreachable**: The site was unreachable from multiple vantage points. The originating proxy contacted at least two peer proxies with at least two packets each, but none elicited a response from the site.
4. **Client Access**: One or more of the originating proxy's access links did not work for resolving DNS names, establishing a TCP session to a server for the Web site, or for contacting any of the peer proxies.
5. **Wide-area**: A link at the originating proxy was working, but the proxy could not use that link either to perform DNS resolution or to contact a server for the desired Web site. Other links and proxies *could* resolve and contact the site, suggesting that the failure was not at either the client access link or the server.

	CSAIL 137,612 sessions			Mazu 9,945 sessions	
Failure Type	**MIT**	**Cog**	**DSL**	**T1**	**Wi**
DNS	1	1	1	1	1
Site RST	50	50	50	2	2
Site Unreach	173	173	173	21	21
Client Access	152	14	2016	0	5
Wide-area	201	238	1828	14	13
Availability	99.6%	99.7%	97%	99.7%	99.6%

Table 3. Observed failures on five Internet links at two sites. The DNS, RST and Unreach rows represent per-site characteristics and are therefore the same for each link at a given proxy.

4.2.1 DNS and Site Failures

After filtering out ten sites with persistent DNS misconfigurations, each proxy observed only one total DNS failure. In both failures, all servers for the domain were on the same LAN. Because DNS resolvers already fail-over after a timeout, MONET's primary benefit is reducing long DNS-related delays.

The 173 site failures in Table 3 show times when no proxy could reach the site but could reach other proxies and other sites. If the proxy received TCP RSTs from the failed site, the server host or program was at fault, not the network. Roughly 20% of the identified site failures sent RSTs to the CSAIL proxy, and 10% sent RSTs to Mazu .

Because of peer proxy restarts and crashes, 8.2% of sessions at the CSAIL proxy never contacted a peer proxy. This analysis thus underestimates the benefits from the overlay, and undercounts the number of site failures by a small margin. We expect to miss about 8.2% (18) of the 223 site failures. In 6.3% (14) instances, MONET could not reach any peers or the site. In our later analysis, most of these instances are probably incorrectly identified as MONET failures instead of unreachable sites. Supporting this conclusion, the proxies observed RSTs from three of the servers in these instances of "MONET failures," similar to the 20% RST rate with the identified server failures. We believe, therefore, there were no instances in which the proxies were unable to reach a functioning site—not surprising, given the number and quality of links involved.

To determine whether this analysis correctly identified failed sites, we re-checked the availability of the unavailable sites two weeks after the first data collection period. 40% of failed sites were *still* unreachable after two weeks. Many of the observed failures were probably attempts to contact per-

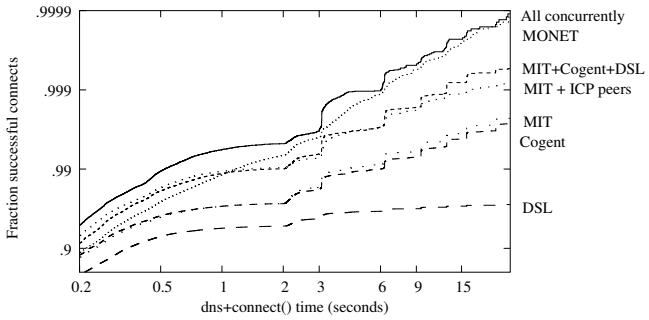


Figure 10. MONET performance at CSAIL. MONET with waypoint selection is nearly as effective as using all paths concurrently, but with only 10% overhead. The MIT+Cogent+DSL and MIT+ICP peers lines use the paths concurrently without waypoint selection delays.

manently failed or non-existent sites.

To better understand how MONET could overcome failures that prevent a client from reaching properly functioning sites, the rest of this analysis excludes positively identified server-side failures. To put these numbers in perspective, Section 4.2.1 examines the (site failure-included) performance of MONET to both all sites, and to a more reliable subset of replicated sites.

4.2.2 Client access link failures

Most links other than the DSL line displayed good availability—near 99.9%. Such high link availability is expected in the environments we measured; for example, MIT (one of the CSAIL proxy's upstream links) is itself connected to the Internet to three upstream ISPs. The remaining unavailability occurred despite the relatively high availability of the links themselves; BGP multi-homing does not provide an end-to-end solution to failures or problems that occur in the middle of the network or close to the server.

We observed one ten-hour failure of the Mazu wireless link during two weeks of monitoring, but it occurred from 9:45pm until 7:45am when little traffic was being replayed through the proxy. The DSL link experienced one 14-hour failure and numerous smaller failures over several months.

We also measured the "global" availability of each link by constantly probing whether or not the link could reach any of the 13 root nameservers. The availability of the links when measured in this fashion is very close to the availability measured through MONET (see [6] for details).

4.3 How Well does MONET Work?

The CSAIL proxy has provided uninterrupted service through 20 major network outages over a 12-month period.[4] One of our most notable results was the ability of a cheap DSL line to improve the availability of the MIT network connection by over an order of magnitude, which we discuss below.

Much of the following analysis concentrates on the effect that MONET has on long delays and failures. To see the overall effects of the proxy, we examine the cumulative distribution of requests whose DNS resolution and `SYN ACK` were received in a certain amount of time, omitting positively identified server failures. Figure 10 shows the "availability" CDFs for MONET and its constituent links at the CSAIL proxy, produced by calculating the fraction of sessions that successfully connected within the time specified by the x-coordinate. This graph and those that follow are in log-scale. The y-axis for the graphs starts near the 90th percentile of connections. The top line, "All concurrently," shows availability when using all paths concurrently, which the proxy performed to gather trace data. A waypoint algorithm simulator picks the order in which MONET's waypoint selection algorithm uses these links, and examines the performance of combinations of the constituent links and peer proxies. MONET's waypoint selection algorithm (Section 2.2) rapidly approaches the "All concurrently" line, and outperforms all of the individual links.

MONET has two effects on availability. First, it reduces exceptional delays. For example, on the Cogent link in Figure 10, 2% of the HTTP sessions require more than 3 seconds to complete DNS resolution and a TCP `connect()`. Combining the MIT link with the Cogent link (which is already one of MIT's upstream ISPs) provides only a small improvement, because packets leaving MIT for many destinations already travel via Cogent. When these links are augmented with a DSL line, however, only 1% of sessions fail to connect within three seconds. The improvements in the 1-3 seconds range are primarily gained by avoiding transient congestion and brief glitches.

The second effect MONET has is improving availability in the face of more persistent failures. Overall, MONET improves availability due to to non-server failures by at least an order of magnitude (*i.e.*, by at least one "nine"). The "MIT+ICP peers" curve in Figure 10 shows that adding remote proxies to a high-uptime link (MIT) can create a more robust system by allowing application-level path selection using existing path diversity. A proxy can realize similar availability benefits by augmenting its primary link with a slower and less reliable DSL line ("MIT+Cogent+DSL"). If a site's primary link is already extremely good, the peer proxy solution increases availability without requiring additional network connectivity, and without periodically directing requests via a much slower DSL line. The benefits of using MONET without local link redundancy will, of course, be limited by the overall availability of the local link. For example, "MIT+ICP peers" achieves 99.92% availability, nearly three times better than the MIT link alone.

4.3.1 Overhead

MONET's waypoint selection algorithm nearly matches the performance of "All concurrently," but adds only 10% more SYNs and 5% more ICP+ packets than a client without MONET. The average Web request (retrieving a single object) handled by our proxy required about 18 packets, so this

additional overhead comes to about 7% of the total packet load, and is a negligible addition to the byte count. The added packets are small—TCP SYN packets are 40 bytes, and the ICP+ query packets are on average around 100 bytes. The mean Web object downloaded through the MIT proxy was 13 kilobytes. The extra SYN and ICP+ packets added by MONET therefore amount to an extra nine bytes per object on average.

The simulation of the waypoint selection algorithm chose a random link to use either 5% or 10% of the time. The benefit of more frequent link probes was at most a 100-200 ms savings in the amount of time it took to find an alternate path when the first path MONET attempted to use had failed—and oftentimes, there was little benefit at all. These latency reductions do not appear to justify the corresponding increase in overhead. A better algorithm might find a better ordering of links for fail-over (*e.g.*, by discovering links whose behavior appears uncorrelated), but because failures are relatively unpredictable, we believe that overcoming transient failures is best done by attempting several alternate links. Waypoint selection avoids links and peers that fail for longer than a few seconds, but does not improve latency in the shorter ranges.

Because of remote proxy failures, random path selection performed poorly. We also simulated MONET using a static retransmit timer instead of using the `rttvar`-derived value. With careful tuning for each proxy, the static value could provide good performance with low overhead, but could not adapt to changing conditions over time.

MONET also introduces overhead from additional DNS lookups. As noted in Section 2.2.2, we believe a MONET with multiple local Internet connections should always send at least two DNS queries. Because DNS queries are frequently cached, the overhead is small—the MONET proxy performed 82,957 DNS lookups to serve 2.1 million objects. The mean packet size for the proxy's DNS queries was 334 bytes. Assuming that the average DNS lookup requires 1.3 packets in each direction [17], duplicating all DNS requests would have added 34 megabytes of traffic over 1.5 months, or 0.1% of the 27.5 gigabytes served by the proxy. Given that between 15 and 27% of queries to the root nameservers are junk queries [17], it is unlikely that the wide deployment of MONET-like techniques would have a negative impact on the DNS infrastructure, particularly since a shared MONET proxy helps aggregate individual lookups through caching.

4.3.2 How well could MONET do?

The top two lines in Figure 10 show the performance of all paths ("All concurrently") and MONET's waypoint selection, respectively. At timescales of 1-2 seconds, the scheme that uses all paths out-performs MONET, because a transient loss or delay forces MONET to wait a few round-trip times before attempting a second connection. Before this time, MONET approximates the performance of its best link; by 1 second, MONET approaches the performance of using two links concurrently.

At longer durations of two to three seconds, MONET comes very close to the performance of all-paths. A part of the difference between these algorithms arises from mispredictions by the waypoint algorithm, and a part probably arises from a conservative choice in our waypoint prediction simulator. The simulator takes the "all paths" data as input, knowing, for instance, that a particular connection attempt took three seconds to complete. The simulator conservatively assumes that the same connection attempt one second later would *also* take three seconds to complete, when in reality it would probably be shorter if the problem were transient.

4.4 Server Failures and Replicated Sites

MONET still improves availability, though less dramatically, in the face of site failures. MONET is more effective at improving availability to replicated or multi-homed sites than to single-homed sites. The leftmost graph in Figure 11 shows the performance of the "all paths" testing with server failures included. This graph includes requests to non-existent servers that could never succeed—the 40% of servers that were *still* unreachable after two weeks—and represents a lower bound on MONET's benefits.

Replicated Web sites, in contrast, generally represent a sample of more available, and presumably well-managed, sites. This category of sites is an imperfect approximation of highly available sites—at least one of the popular multi-homed sites in our trace exhibited recurring server failures—but as the data illustrated in Figure 11 shows, these sites do exhibit generally higher availability than the average site. The replicated services we measured typically used combinations of clustering, BGP multi-homing, and low-TTL DNS redirection to direct clients to functioning servers.[5]

23,092 (17%) of the sessions we observed went to Web sites that advertised multiple IP addresses. Web sessions to these multiple-address sites are dominated by Content Delivery Networks (CDNs) and large content providers. For example, Akamai, Speedera, the New York Times, and CNN account for 53% of the sessions.

Intriguingly, a single link's access to the multiply announced subset of sites is *not* appreciably more reliable than accesses to all sites (Figure 11, right). The MIT connection achieved 99.4% reachability to all sites, and 99.5% to the multi-homed site subset. When augmented with peer proxies, the MIT connection achieved 99.8% availability. Using all local interfaces, MONET achieved 99.92%, and reached 99.93% after just *six seconds* when using both its local interfaces and peer proxies.

MONET's reduction of network failures is more apparent when communicating with replicated sites. The improved performance in accessing these sites shows that MONET's use of multiple server addresses is effective, and that MONET's techniques complement CDN-like replication to further improve availability.

The foregoing analysis counted as replicated *all* sites that

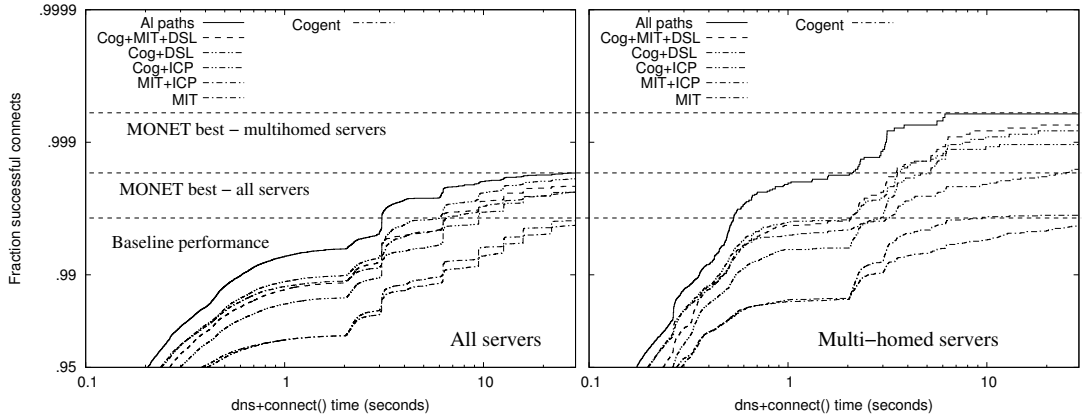


Figure 11. HTTP session success, including server failures, through the CSAIL proxy to all servers (left) and only multi-homed servers (right). The success rate of the base links is unchanged, but MONETs effectiveness is enhanced when contacting multi-homed services.

advertised multiple IP addresses. We assigned sites to a content provider by a breadth-first traversal of the graph linking hostnames to IP addresses, creating clusters of hosted sites. We manually identified the content provider for the largest clusters and created regular expressions to match other likely hosts within the same provider. These heuristics identified 1,649 distinct IP addresses belonging to 38 different providers. While this method will not wrongly assign a request to a content provider, it is not guaranteed to find *all* of the requests sent to a particular provider.

4.5 Discussion and Limitations

MONET masked numerous major failures at the borders of its host networks and in the wide-area. In the cable modem deployment, its ability to balance load between multiple access links provided appreciable performance gains. MONET's benefits are, however, subject to several limitations, some fundamental and some tied to the current implementation:

Site failures: Two power failures at the CSAIL proxy created failures that MONET could not overcome.[6] Improvements provided by the proxy are bounded by the limitations of its environment, which may represent a more significant obstacle than the network in some deployments.

Probes do not always determine success: A failed Internet2 router near MIT's border began dropping most packets larger than 400 bytes. Because MONET uses small ($\sim$ 60 byte) SYN packets to probe paths, the proxy was ineffective against this bizarre failure. While MONET's probes are more "end-to-end" than the checks provided by other systems, there are failures that could be specifically crafted to defeat a MONET-like system. A higher-level progress check that monitored whether or not data was still flowing on an HTTP connection could provide resilience to some of these failures and to mid-stream failures by re-issuing the HTTP request if necessary. Such solutions must avoid undesirable side-effects such as re-issuing a credit card purchase.

Software failures. Several Web sites could never be reached directly, but could always be contacted through a remote proxy. These sites sent invalid DNS responses that were accepted by the BIND 8 name server running on the remote proxies, but that were discarded by the BIND 9 nameserver on the multi-homed proxies. While these anomalies were rare, affecting only two of the Web sites accessed through the MIT proxy, they show some benefits from having diversity in software implementations in addition to diversity in physical and network paths.

Download times. Initial connection latency is a critical factor for interactive Web sessions. Total download time, however, is more important for large transfers. Earlier studies suggest that connection latency is effective in server selection [11], but there is no guarantee that a successful connection indicates a low-loss path. We briefly tested whether this held true for the MONET proxies. A client on the CSAIL proxy fetched one of 12,181 URLs through two randomly chosen paths at a time to compare their download times, repeating this process 240,000 times over 36 days. The SYN response time correctly predicted the full HTTP transfer 83.5% of the time. The objects fetched were a random sample from the static objects downloaded by users of our proxy.

5 Related Work

Benefits from path choice. The RON [7], Detour [31] and Akarouting [23] studies demonstrated that providing clients with a choice of paths to the server increases both peformance and reliability. The RON study found that single-hop overlay routing provided most of the benefits achievable by overlay routing. The recent SOSR work expanded upon these findings, showing that selecting just four random intermediaries provided excellent reliability with low overhead [15]. The SOSR study focused on failures lasting between 30 seconds to six minutes; the MONET results suggest that the SOSR results also apply at shorter time-scales.

Akella *et al.* found that multi-homing two local links us-

ing route control can improve latency by about 25% [2]. The improvements are insensitive to the exact route control mechanism and measurement algorithms [4]. These results complement our findings: MONET focuses primarily on strategies for achieving the reliability benefits of multi-homing (the worst 5 percent of responses), while these studies focus on latency improvements.

Their more recent study of five days of pings between 68 Internet nodes found that most paths have an availability of around 99.9% [3]. These numbers are consistent with our estimates of link failure rates; the remainder of our breakdown analyzes the contribution of other sources of failure and extends this analysis to a much wider set of hosts.

Commercial products like Stonesoft's "Multi-Link Technology" send multiple TCP SYNs to servers to multi-home clients without BGP [36]. RadWare's "LinkProof" pings a small set of external addresses to monitor connectivity on each link, failing over if a link appears down [14]. These systems, and others, can help balance load across multiple links [16]

The Smart Clients approach downloads mobile code to clients, providing flexible and effective server selection. [40]. MONET achieves many of the same reliability benefits without changes to name resolution and without mobile code.

Content Delivery Networks (CDNs) such as Akamai [1] and CoDeen [27] use DNS, server redirects, and client proxy configuration to redirect clients to intermediate nodes, which cache content for quicker access. CDNs deliver replicated popular content and are particularly effective in the face of flash crowds [34, 35], but, without additional reliability mechanisms like those discussed in this paper, are not as effective against network disruptions to un-cached content and access link failures. In fact, our results showed that MONET can improve the performance of CDN-hosted sites.

CoDNS [28] masks DNS lookup delays by proxying DNS requests through peers. When CoDNS does not hear a DNS response from its local nameserver within a short static timeout (200 to 500ms, typically), the CoDNS resolver forwards the query to a peer node. When a majority of recent requests get resolved through a peer node, CoDNS instead immediately sends all queries both locally and through the peer.

Multi-homing Techniques. BGP-based techniques recover only from link failures, and require a few minutes to do so [20]. BGP's route aggregation suppresses the announcement of failures within an aggregate, financial and technical requirements preclude many small clients from using it. These limitations are partly addressed by traffic control systems and higher-layer multi-homing techniques.

RouteScience [30] and SockEye [32] use end-to-end measurements to select outbound routes for networks with multiple BGP-speaking Internet links. To control the inbound link, the following systems change the IP address from which traffic originates, forcing traffic to return to one machine augmented with multiple Internet connections or to a specific overlay node. SOSR, Detour, and NATRON [39] all interpose a NAT on outbound traffic; MONET uses an application-layer proxy. While NAT is more general, the MONET proxy provides more information and is easier to partially deploy (Section 2.3). All of these approaches change the outbound IP address in some fairly intrusive way.

6 Conclusion

This paper presented MONET, a Web proxy system to improve the end-to-end client-perceived availability of accesses to Web sites. MONET masks several kinds of failures that prevent clients from connecting to Web sites, including access link failures, Internet routing failures, DNS failures, and a subset of server-side failures. MONET masks these failures by obtaining and exploring multiple paths between the proxy and Web sites, considering paths via its multi-homed local links, via peer MONET proxies, and to multiple server IP addresses. MONET incorporates a waypoint selection algorithm that allows a proxy to explore these different paths with little overhead, while also achieving quick failure recovery, usually within a few round-trip times.

In contrast to approaches that improve a specific component of the end-to-end path from Web client to server, MONET incorporates simple, reusable failure-masking techniques that overcome failures in many different components.

We deployed a single-proxy multi-homed MONET two years ago. The version of the system described in this paper using multiple proxies has been operational for over 18 months, and has been in daily use by a user community of at least fifty users. The MONET code is publicly available.

Our experimental analysis of traces from a real-world MONET deployment show that MONET corrected nearly *all* observed failures where the server (or the server access network) itself had not failed. MONET's simple waypoint selection algorithm performs almost as well as an "omniscient" scheme that sends requests on all available interfaces. In practice, for a modest overhead of 0.1% (bytes) and 6% (packets), we find that between 60% and 94% of all observed failures can be eliminated (on the different measured physical links), and the "number of nines" of non-server-failed availability can be improved by one to two nines.

Our experience with MONET suggests that Web access availability can be improved by an order of magnitude or more using an inexpensive and relatively low speed link (*e.g.*, a DSL link), or using a few other peer proxies. The techniques incorporated in MONET demonstrate that the cost of high Web access availability (three to four "nines") need not be daunting.

We believe that MONET's end-to-end approach addresses all the reasons for service unavailability and our experimental results show that these failures are maskable, except

for server failures themselves. With MONET in place, the main remaining barrier to "five nines" or better availability is server-side failure resilience.

Acknowledgments

The NSDI reviewers and several of our colleagues provided great feedback: Nick Feamster, Mary Lou Godbe, John Guttag, Max Krohn, Stan Rost, Jon Salz, Mythili Vutukuru, and Michael Walfish. Special thanks to our "shepherd," David Wetherall, who helped us look at the paper with fresh eyes after too much editing. We appreciate the (necessarily anonymous) people at MIT who used our proxies for so long to provide us with measurement data, and the administrators who hosted our proxies. This work was funded by the NSF under Cooperative Agreement No. ANI-0225660; David Andersen was funded by a Microsoft Research fellowship.

References

[1] Akamai. http://www.akamai.com, 1999.

[2] AKELLA, A., MAGGS, B., SESHAN, S., SHAIKH, A., AND SITARAMAN, R. A measurement-based analysis of multihoming. In *Proc. ACM SIGCOMM* (Karlsruhe, Germany, Aug. 2003).

[3] AKELLA, A., PANG, J., MAGGS, B., SESHAN, S., AND SHAIKH, A. A comparison of overlay routing and multihoming route control. In *Proc. ACM SIGCOMM* (Portland, OR, Aug. 2004).

[4] AKELLA, A., SESHAN, S., AND SHAIKH, A. Multihoming performance benefits: An experimental evaluation of practical enterprise strategies. In *Proc. USENIX Annual Technical Conference* (Boston, MA, June 2004).

[5] ALBITZ, P., AND CIU, C. *DNS and BIND in a Nutshell*. O'Reilly and Associates, 1992.

[6] ANDERSEN, D. G. *Improving End-to-End Availability Using Overlay Networks*. PhD thesis, Massachusetts Institute of Technology, Feb. 2005.

[7] ANDERSEN, D. G., BALAKRISHNAN, H., KAASHOEK, M. F., AND MORRIS, R. Resilient Overlay Networks. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)* (Banff, Canada, Oct. 2001), pp. 131–145.

[8] BERNSTEIN, D. J. SYN Cookies. http://cr.yp.to/syncookies.html, 1996.

[9] COHEN, E., AND KAPLAN, H. Prefetching the means for document transfer: A new approach for reducing Web latency. In *Proc. IEEE INFOCOM* (Tel-Aviv, Israel, Mar. 2000), vol. 2, pp. 854–863.

[10] DAHLIN, M., CHANDRA, B., GAO, L., AND NAYATE, A. End-to-end WAN Service Availability. *IEEE/ACM Transactions on Networking 11*, 2 (Apr. 2003).

[11] DYKES, S. G., ROBBINS, K. A., AND JEFFERY, C. L. An empirical evaluation of client-side server selection algorithms. In *Proc. IEEE INFOCOM* (Tel-Aviv, Israel, Mar. 2000), vol. 3, pp. 1361–1370.

[12] ELZ, R., BUSH, R., BRADNER, S., AND PATTON, M. *Selection and Operation of Secondary DNS Servers*. Internet Engineering Task Force, July 1997. RFC 2182 / BCP 16.

[13] ENRIQUEZ, P., BROWN, A., AND PATTERSON, D. A. Lessons from the PSTN for dependable computing. In *Workshop on Self-Healing, Adaptive and Self-Managed Systems* (2002).

[14] GREENFIELD, D. RadWare Linkproof. *Network Magazine* (Dec. 1999). `http://www.networkmagazine.com/article/NMG20000427S0005`.

[15] GUMMADI, K. P., MADHYASTHA, H. V., GRIBBLE, S. D., LEVY, H. M., AND WETHERALL, D. Improving the reliability of Internet paths with one-hop source routing. In *Proc. 6th USENIX OSDI* (San Francisco, CA, Dec. 2004).

[16] GUO, F., CHEN, J., LI, W., AND CKER CHIUEH, T. Experiences in building a multihoming load balancing system. In *Proc. IEEE INFOCOM* (Hong Kong, Mar. 2004).

[17] JUNG, J., SIT, E., BALAKRISHNAN, H., AND MORRIS, R. DNS Performance and the Effectiveness of Caching. In *Proc. ACM SIGCOMM Internet Measurement Workshop* (San Fransisco, CA, Nov. 2001).

[18] KRISHNAMURTHY, B., AND REXFORD, J. *Web Protocols and Practice*. Addison Wesley, Upper Saddle River, NJ, 2001.

[19] KUHN, D. R. Sources of failure in the public switched telephone network. *IEEE Computer 30*, 4 (1997).

[20] LABOVITZ, C., AHUJA, A., BOSE, A., AND JAHANIAN, F. Delayed Internet Routing Convergence. In *Proc. ACM SIGCOMM* (Stockholm, Sweden, Sept. 2000), pp. 175–187.

[21] LEMON, J. Resisting SYN flood DoS attacks with a SYN cache. In *BSDCon* (Feb. 2002).

[22] MAHAJAN, R., WETHERALL, D., AND ANDERSON, T. Understanding BGP misconfiguration. In *Proc. ACM SIGCOMM* (Pittsburgh, PA, Aug. 2002), pp. 3–17.

[23] MILLER, G. Overlay routing networks (Akarouting). `http://www-math.mit.edu/~steng/18.996/lecture9.ps`, Apr. 2002.

[24] MOESTL, T., AND ROMBOUTS, P. pdnsd. `http://www.phys.uu.nl/~rombouts/pdnsd.html`, 2004.

[25] NETWORK RELIABILITY AND INTEROPERABILITY COUNCIL (NRIC). Network reliability: A report to the nation, 1993. `http://www.nric.org/pubs/nric1/index.html`.

[26] NETWORK RELIABILITY AND INTEROPERABILITY COUNCIL (NRIC). Network reliability—the path forward, 1996. http://www.nric.org/pubs/nric2/fg1/index.html.

[27] PAI, V. S., WANG, L., PARK, K., PANG, R., AND PETERSON, L. The dark side of the Web: An open proxy's view. In *Proc. 2nd ACM Workshop on Hot Topics in Networks (Hotnets-II)* (Cambridge, MA, Nov. 2003).

[28] PARK, K., PAI, V., PETERSON, L., AND WANG, Z. CoDNS: Improving DNS performance and reliability via cooperative lookups. In *Proc. 6th USENIX OSDI* (San Francisco, CA, Dec. 2004).

[29] PAXSON, V. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD thesis, U. C. Berkeley, May 1997.

[30] RouteScience. Whitepaper available from `http://www.routescience.com/technology/tec_whitepaper.html`.

[31] SAVAGE, S., COLLINS, A., HOFFMAN, E., SNELL, J., AND ANDERSON, T. The End-to-End Effects of Internet Path Selection. In *Proc. ACM SIGCOMM* (Cambridge, MA, Sept. 1999), pp. 289–299.

[32] Sockeye. `http://www.sockeye.com/`.

[33] Squid Web Proxy Cache. `http://www.squid-cache.org/`.

[34] STADING, T., MANIATIS, P., AND BAKER, M. Peer-to-peer caching schemes to address flash crowds. In *Proc. 1st International Workshop on Peer-to-Peer Systems (IPTPS)* (Cambridge, MA, Mar. 2002).

[35] STAVROU, A., RUBENSTEIN, D., , AND SAHU, S. A lightweight, robust P2P system to handle flash crowds. In *IEEE International Conference on Network Protocols (ICNP)* (Paris, France, Nov. 2002).

[36] STONESOFT. Multi-link technology white paper. `http://www.stonesoft.com/files/products/StoneGate/SG_Multi-Link_Technol%ogy_Whitepaper.pdf`, Oct. 2001.

[37] VIXIE, P., AND WESSELS, D. *Hyper Text Caching Protocol (HTCP/0.0)*. Internet Engineering Task Force, Jan. 2000. RFC 2756.

[38] WESSELS, D., AND CLAFFY, K. *Application of Internet Cache Protocol (ICP), version 2*. Internet Engineering Task Force, Sept. 1997. RFC 2187.

[39] YIP, A. NATRON: Overlay routing to oblivious destinations. Master's thesis, MIT, Aug. 2002.

[40] YOSHIKAWA, C., CHUN, B., EASTHAM, P., VAHDAT, A., ANDERSON, T., AND CULLER, D. Using smart clients to build scalable services. In *Proc. USENIX Annual Technical Conference* (Anaheim, CA, Jan. 1997).

[41] ZONA RESEARCH. The need for speed II. *Zona Market Bulletin 5* (Apr. 2001). `http://www.keynote.com/downloads/Zona_Need_For_Speed.pdf`.

Notes

[1] Caching complements MONET's path selection by improving the delivery of "hot" static objects, but caching alone does not improve availability [10]. In contrast, MONET's path selection also improves access to uncacheable dynamic content such as that found in Web-based commerce applications.

[2] The gaps between requests from a single user to one Web site are usually under 60 seconds ([18], pp. 394

[3] Several sites had persistent DNS lame delegations; another site always returned `0.0.0.0` as its IP address. One site returned DNS names that exceeded the 128 byte capture length used to obtain the DNS packets.

[4] During the first outage it experienced, we realized that the proxy failed to perform redundant DNS lookups; fixing this shortcoming permitted uninterrupted service during all known outages thereafter. Many of these early outages occurred before our detailed measurement period.

[5] This analysis assumes that being able to reach a replica denotes service availability, which may or may not be the case with some caching CDNs.

[6] It is likely that most of the clients also had power failures, but clients accessing the proxy from other buildings may have been affected.

Shark: Scaling File Servers via Cooperative Caching

Siddhartha Annapureddy, Michael J. Freedman, David Mazières
New York University
http://www.scs.cs.nyu.edu/shark/

Abstract

Network file systems offer a powerful, transparent interface for accessing remote data. Unfortunately, in current network file systems like NFS, clients fetch data from a central file server, inherently limiting the system's ability to scale to many clients. While recent distributed (peer-to-peer) systems have managed to eliminate this scalability bottleneck, they are often exceedingly complex and provide non-standard models for administration and accountability. We present Shark, a novel system that retains the best of both worlds—the scalability of distributed systems with the simplicity of central servers.

Shark is a distributed file system designed for large-scale, wide-area deployment, while also providing a drop-in replacement for local-area file systems. Shark introduces a novel cooperative-caching mechanism, in which mutually-distrustful clients can exploit each others' file caches to reduce load on an origin file server. Using a distributed index, Shark clients find nearby copies of data, even when files originate from different servers. Performance results show that Shark can greatly reduce server load and improve client latency for read-heavy workloads both in the wide and local areas, while still remaining competitive for single clients in the local area. Thus, Shark enables modestly-provisioned file servers to scale to hundreds of read-mostly clients while retaining traditional usability, consistency, security, and accountability.

1 Introduction

Users of distributed computing environments often launch similar processes on hundreds of machines nearly simultaneously. Running jobs in such an environment can be significantly more complicated, both because of data-staging concerns and the increased difficulty of debugging. Batch-oriented tools, such as Condor [9], can provide I/O transparency to help distribute CPU-intensive applications. However, these tools are ill-suited to tasks like distributed web hosting and network measurement, in which software needs low-level control of network functions and resource allocation. An alternative is frequently seen on network test-beds such as RON [2] and PlanetLab [24]: users replicate their programs, along with some minimal execution environment, on every machine before launching a distributed application.

Replicating execution environments has a number of drawbacks. First, it wastes resources, particularly bandwidth. Popular file synchronization tools do not optimize for network locality, and they can push many copies of the same file across slow network links. Moreover, in a shared environment, multiple users will inevitably copy the exact same files, such as popular OS add-on packages with language interpreters or shared libraries. Second, replicating run-time environments requires hard state, a scarce resource in a shared test-bed. Programs need sufficient disk space, yet idle environments continue to consume disk space, in part because the owners are loathe to consume the bandwidth and effort required for redistribution. Third, replicated run-time environments differ significantly from an application's development environment, in part to conserve bandwidth and disk space. For instance, users usually distribute only stripped binaries, not source or development tools, making it difficult to debug running processes in a distributed system.

Shark is a network file system specifically designed to support widely distributed applications. Rather than manually replicate program files, users can place a distributed application and its entire run-time environment in an exported file system, and simply execute the program directly from the file system on all nodes. In a chrooted environment such as PlanetLab, users can even make `/usr/local` a symbolic link to a Shark file system, thereby trivially making all local software available on all test-bed machines.

Of course, the big challenge faced by Shark is scalability. With a normal network file system, if hundreds of clients suddenly execute a large, 40MB C++ program from a file server, the server quickly saturates its network uplink and delivers unacceptable performance. Shark, however, scales to large numbers of clients through a locality-aware cooperative cache. When reading an uncached file, a Shark client avoids transferring the file or even chunks of the file from the server, if the same data can be fetched from another, preferably nearby, client. For world-readable files, clients will even download nearby

cached copies of identical files—or even file chunks—originating from different servers.

Shark leverages a locality-aware, peer-to-peer distributed index [10] to coordinate client caching. Shark clients form self-organizing clusters of well-connected machines. When multiple clients attempt to read identical data, these clients locate nearby replicas and stripe downloads from each other in parallel. Thus, even modestly-provisioned file servers can scale to hundreds, possibly thousands, of clients making mostly read accesses.

There have been serverless, peer-to-peer file systems capable of scaling to large numbers of clients, notably Ivy [23]. Unfortunately, these systems have highly non-standard models for administration, accountability, and consistency. For example, Ivy spreads hard state over multiple machines, chosen based on file system data structure hashes. This leaves no single entity ultimately responsible for the persistence of a given file. Moreover, peer-to-peer file systems are typically noticeably slower than conventional network file systems. Thus, in both accountability and performance they do not provide a substitute for conventional file systems. Shark, by contrast, exports a traditional file-system interface, is compatible with existing backup and restore procedures, provides competitive performance on the local area network, and yet easily scales to many clients in the wide area.

For workloads with no read sharing between users, Shark offers performance that is competitive with traditional network file systems. However, for shared read-heavy workloads in the wide area, Shark greatly reduces server load and improves client latency. Compared to both NFSv3 [6] and SFS [21], a secure network file system, Shark can reduce server bandwidth usage by nearly an order of magnitude and can provide a 4x-6x improvement for client latency for reading large files, as shown by both local-area experiments on Emulab wide-area experiments on the PlanetLab test-bed.

By providing scalability, efficiency, and security, Shark enables network file systems to be employed in environments where they were previously impractical. Yet Shark retains their attractive API, semantics, and portability: Shark interacts with the local host using an existing network file system protocol (NFSv3) and runs in user space.

The remainder of this paper in organized as follows. Section 2 details the design of Shark: its file-system components, caching and security protocols, and distributed index operations. Section 3 describes its implementation, and Section 4 evaluates Shark's performance. Section 5 discusses related work, and Section 6 concludes.

2 Shark Design

Shark's design incorporates a number of key ideas aimed at reducing the load on the server and improving client-perceived latencies. Shark enables clients to securely mount remote file systems and efficiently access them. When a client reads a particular file first, it fetches the data from the file server. Upon retrieving the file, the client caches it and registers itself as a *replica proxy* (or *proxy* for short) for the "chunks" of the file in the distributed index. Subsequently, when another client attempts to access the file, it discovers proxies for the file chunks by querying the distributed index. The client then establishes a secure channel to multiple such proxies and downloads the file chunks in parallel. (Note that the client and the proxy are mutually distrustful.) Upon fetching these chunks, the client registers itself also as a proxy for these chunks.

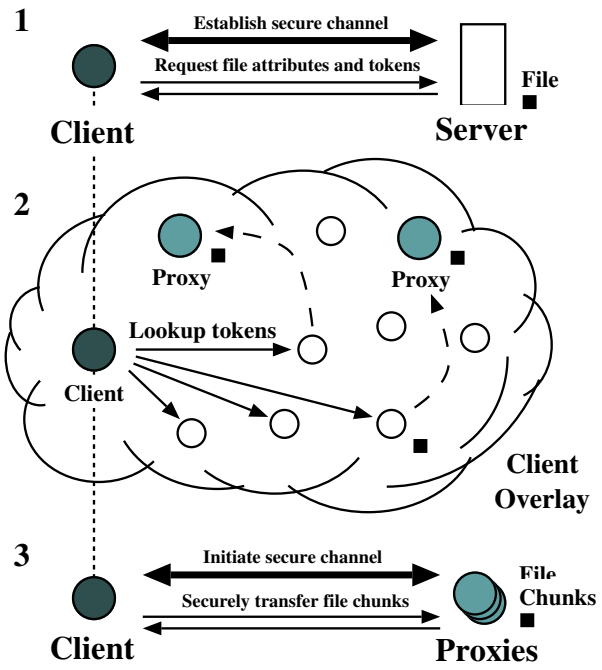


Figure 1: *Shark System Overview.* A client machine simultaneously acts as a client (to handle local application file system accesses), as a proxy (to serve cached data to other clients), and as a node (within the distributed index overlay). In a real deployment, there may be multiple file servers that each host separate file systems, and each client may access multiple file systems. For simplicity, however, we show a single file server.

Figure 1 provides an overview of the Shark system. First, when a client attempts to read a file, it queries its file server for the file's attributes and some opaque tokens (Step 1 as shown). One token identifies the contents of the whole file, while other tokens each identify a particular *chunk* of the file. A Shark server divides a file into chunks by running a Rabin fingerprint algorithm on the file [22]. This technique splits a file along specially chosen boundaries in such a way that preserves data common-

alities across files, for example, between file versions or when concatenating files.

Next, a client attempts to discover replica proxies for the particular file via Shark's distributed index (Step 2). Shark clients organize themselves into a key/value indexing infrastructure, built atop a peer-to-peer structured routing overlay [10]. For now, we can visualize this layer as exposing two operations, put and get: A client executes put to declare that it has something; get returns the list of clients who have something. A Shark client uses its tokens to derive *indexing keys* that serve as inputs to these operations. It uses this distributed index to register itself and to find other nearby proxies caching a file chunk.

Finally, a client connects to several of these proxies, and it requests various chunks of data from each proxy in parallel (Step 3). Note, however, that clients themselves are mutually distrustful, so Shark must provide various mechanisms to guarantee secure data sharing: (1) Data should be encrypted to preserve confidentiality and should be decrypted only by those with appropriate read permissions. (2) A malicious proxy should not be able to break data integrity by modifying content without a client detecting the change. (3) A client should not be able to download large amounts of even encrypted data without proper read authorization.

Shark uses the opaque tokens generated by the file server in several ways to handle these security issues. (1) The tokens serve as a shared secret (between client and proxy) with which to derive symmetric cryptographic keys for transmitting data from proxy to client. (2) The client can verify the integrity of retrieved data, as the token acts to bind the file contents to a specific verifiable value. (3) A client can "prove" knowledge of the token to a proxy and thus establish read permissions for the file. Note that the indexing keys used as input to the distributed index are only derived from the token; they do not in fact expose the token's value and thus otherwise destroy its usefulness as a shared secret.

Shark allows clients to share common data segments on a sub-file granularity. As a file server provides the tokens naming individual file chunks, clients can share data at the granularity of chunks as opposed to whole files.

In fact, Shark provides *cross-file-system sharing* when tokens are derived solely from file contents. Consider the case when users attempt to mount `/usr/local` (for the same Operating System) using different file servers. Most of the files in these directories are identical and even when the file versions are different, many of the chunks are identical. Thus, even when distinct subsets of clients access different file servers to retrieve tokens, one can still act as a proxy for the other to transmit the data.

In this section, we first describe the Shark file server (Section 2.1), then discuss the file consistency provided by Shark (2.2). Section 2.3 describes Shark's cooperative caching, its cryptographic operations, and client-proxy protocols. Finally, we present Shark's chunking algorithm (2.4) and its distributed index (2.5) in more depth.

2.1 Shark file servers

Shark names file systems using self-certifying pathnames, as in SFS [21]. These pathnames *explicitly* specify all information necessary to securely communicate with remote servers. Every Shark file system is accessible under a pathname of the form:

$$/\texttt{shark}/@server, pubkey$$

A Shark server exports local file systems to remote clients by acting as an NFS loop-back client. A Shark client provides access to a remote file system by automounting requested directories [21]. This allows a client-side Shark NFS loop-back server to provide unmodified applications with seamless access to remote Shark file systems. Unlike NFS, however, all communication with the file server is sent over a secure channel, as the self-certifying pathname includes sufficient information to establish a secure channel.

System administrators manage a Shark server identically to an NFS server. They can perform backups, manage access controls with little difference. They can configure the machine to taste, enforce various policies, perform security audits etc. with *e*xisting tools. Thus, Shark provides system administrators with a familiar environment and thus can be deployed painlessly.

2.2 File consistency

Shark uses two network file system techniques to improve read performance and decrease server load: leases [11] and AFS-style whole-file caching [14]. When a user attempts to read any portion of a file, the client first checks its disk cache. If the file is not already cached or the cached copy is not up to date, the client fetches a new version from Shark (either from the cooperative cache or directly from the file server).

Whenever a client makes a read RPC to the file server, it gets a read lease on that particular file. This lease corresponds to a commitment from the server to notify the client of any modifications to the file within the lease's duration. Shark uses a default lease duration of five minutes. Thus, if a user attempts to reads from a file—and if the file is cached, its lease is not expired, and no server notification (or *callback*) has been received—the read succeeds immediately using the cached copy.

If the lease has already expired when the user attempts to read the file, the client contacts the file server for fresh file attributes. The attributes, which include file permissions, mode, size, etc., also provide the file's modification

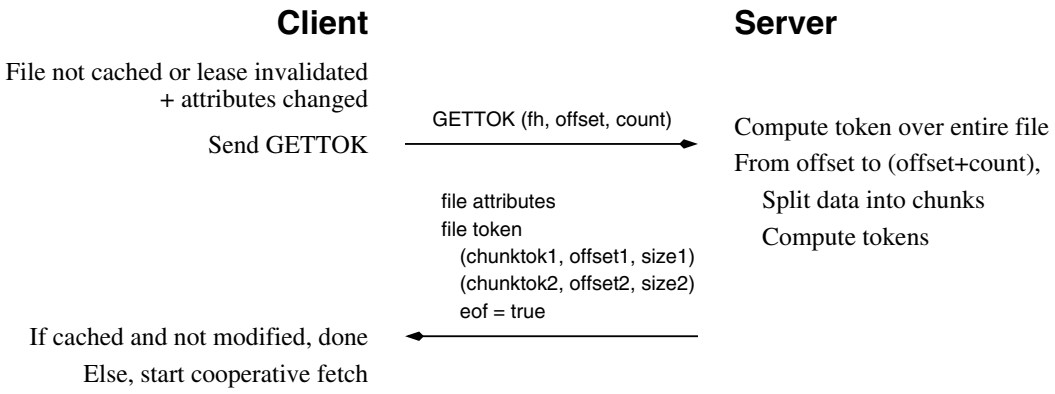


Figure 2: Shark GETTOK RPC

and inode change times. If these times are the same as the cached copy, no further action is necessary: the cached copy is fresh and the client renews its lease. Otherwise, the client needs to fetch a new version from Shark.

While these techniques reduce unnecessary data transfers when files have not been modified, each client needs to refetch the entire file after any modification from the server. Thus, large numbers of clients for a particular file system may overload the server and offer poor performance. Two techniques alleviate the problem: Shark fetches only modified chunks of a file, while its cooperative caching allows clients to fetch data from each other instead of from the server.

While Shark attempts to handle reads within its cooperative cache, all writes are sent to the origin server. When any type of modification occurs, the server must invalidate all unexpired leases, update file attributes, recompute its file token, and update its chunk tokens and boundaries.

We note that a reader can get a mix of old and new file data if a file is modified *while* the reader is fetching file attributes and tokens from the server. (This condition can occur when fetching file tokens requires multiple RPCs, as described next.) However, this behavior is no different from NFS, but it could be changes using AFS-style whole-file overwrites [14].

2.3 Cooperative caching

File reads in Shark make use of one RPC procedure not in the NFS protocol, GETTOK, as shown in Figure 2.

GETTOK supplies a file handle, offset, and count as arguments, just as in a READ RPC. However, instead of returning the actual file data, it returns the file's attributes, the *file token*, and a vector of *chunk descriptions*. Each chunk description identifies a specific extent of the file by offset and size, and includes a *chunk token* for that extent. The server will only return up to 1,024 chunk descriptions in one GETTOK call; the client must issue multiple calls for larger files.

The file attributes returned by GETTOK include sufficient information to determine if a local cached copy is up-to-date (as discussed). The tokens allow a client (1) to discover current proxies for the data, (2) to demonstrate read permission for the data to proxies, and (3) to verify the integrity of data retrieved from proxies. First, let us specify how Shark's various tokens and keys are derived.

Content-based naming. Shark names content with cryptographic hash operations, as given in Table 1.

A *file token* is a 160-bit value generated by a cryptographic hash of the file's contents F and some optional per-file randomness r that a server may use as a key for each file (discussed later):

$$T_F = tok(F) = \mathsf{HMAC}_r(F)$$

Throughout our design, HMAC is a keyed hash function [4], which we instantiate with SHA-1. We assume that SHA-1 acts as a collision-resistant hash function, which implies that an adversary cannot find an alternate input pair that yields the same T_F.[1]

The *chunk token* T_{F_i} in a chunk description is also computed in the same manner, but only uses the particular chunk of data (and optional randomness) as an input to SHA-1, instead of the entire file F. As file and chunk tokens play similar roles in the system, we use T to refer to either type of token indiscriminately.

The *indexing key* I used in Shark's distributed index is simply computed by $\mathsf{HMAC}_T(\mathsf{I})$. We key the HMAC function with T and include a special character I to signify indexing. More specifically, I_F refers to the indexing key for file F, and I_{F_i} for chunk F_i.

The use of such server-selected randomness r ensures that an adversary cannot guess file contents, given only I. Otherwise, if the file is small or stylized, an adversary

[1] While our current implementation uses SHA-1, we could similarly instantiate HMAC with SHA-256 for greater security.

Symbol	Description	Generated by ...	Only known by ...
F	File		Server and approved readers
F_i	ith file chunk	Chunking algorithm	Parties with access to F
r	Server-specific randomness	$r = \mathsf{PRNG}()$ or $r = 0$	Parties with access to F
T	File/chunk token	$tok(F) = \mathsf{HMAC}_r(F)$	Parties with access to F/F_i
$\mathsf{I}, \mathsf{E}, \mathsf{A}_C, \mathsf{A}_P$	Special constants	System-wide parameters	Public
I	Indexing key	$\mathsf{HMAC}_T(\mathsf{I})$	Public
r_C, r_P	Session nonces	$r_C, r_P = \mathsf{PRNG}()$	Parties exchanging F/F_i
$Auth_C$	Client authentication token	$\mathsf{HMAC}_T(\mathsf{A}_C, C, P, r_C, r_P)$	Parties exchanging F/F_i
$Auth_P$	Proxy authentication token	$\mathsf{HMAC}_T(\mathsf{A}_P, P, P, r_P, r_C)$	Parties exchanging F/F_i
K_E	Encryption key	$\mathsf{HMAC}_T(\mathsf{E}, C, P, r_C, r_P)$	Parties exchanging F/F_i

Table 1: Notation used for Shark values

may be able to perform an offline brute-force attack by enumerating all possibilities.

On the flip-side, omitting this randomness enables cross-file-system sharing, as its content-based naming can be made independent of the file server. That is, when r is omitted and replaced by a string of 0s, the distributed indexing key is dependent only on the contents of F: $I_F = \mathsf{HMAC}_{\mathsf{HMAC}_0(F)}(\mathsf{I})$. Cross-file-system sharing can improve client performance and server scalability when nearby clients use different servers. Thus, the system allows one to trade-off additional security guarantees with potential performance improvements. By default, we omit this randomness for world-readable files, although configuration options can override this behavior.

The cooperative-caching read protocol. We now specify in detail the cooperative-caching protocol used by Shark. The main goals of the protocol are to reduce the load on the server and to improve client-perceived latencies. To this end, a client tries to download chunks of a file from multiple proxies in parallel. At a high level, a client first fetches the tokens for the chunks that comprise a file. It then contacts nearby proxies holding each chunk (if such proxies exists) and downloads them accordingly. If no other proxy is caching a particular chunk of interest, the client falls back on the server for that chunk.

The client sends a GETTOK RPC to the server and fetches the whole-file token, the chunk tokens, and the file's attributes. It then checks its cache to determine whether it has a fresh local copy of the file. If not, the client runs the following cooperative read protocol.

The client always attempts to fetch k chunks in parallel. We can visualize the client as spawning k threads, with each thread responsible for fetching its assigned chunk.[2] Each thread is assigned a *random* chunk F_i from the list of needed chunks. The thread attempts to discover nearby proxies caching that chunk by querying the distributed index using the primitive $get(I_{F_i} = \mathsf{HMAC}_{T_{F_i}}(\mathsf{I}))$. If this *get* request fails to find a proxy or does not find one within a specified time, the client fetches the chunk from the server. After downloading the entire chunk, the client announces itself in the distributed index as a proxy for F_i.

If the get request returns several proxies for chunk F_i, the client chooses one with minimal latency and establishes a secure channel with the proxy, as described later. If the security protocol fails (perhaps due to a malicious proxy), the connection to the proxy fails, or a newly specified time is exceeded, the thread chooses another proxy from which to download chunk F_i. Upon downloading F_i, the client verifies its integrity by checking whether $T_{F_i} \stackrel{?}{=} tok(F_i)$. If the client fails to successfully download F_i from any proxy after a fixed number of attempts, it falls back onto the origin file server.

Reusing proxy connections. While a client is downloading a chunk from a proxy, it attempts to reuse the connection to the proxy by *negotiating* for other chunks. The client picks α random chunks still needed. It computes the corresponding α indexing keys and sends these to the proxy. The proxy responds with those a chunks, among the α requested, that it already has. If $a = 0$, the proxy responds instead with β keys corresponding to chunks that it does have. The client, upon downloading the current chunk, selects a new chunk from among those negotiated (*i.e.*, needed by the client and known to the proxy). The client then proves read permissions on the new chunk and begins fetching the new chunk. If no such chunks can be negotiated, the client terminates the connection.

Client-proxy interactions. We now describe the secure communication mechanisms between clients and proxies that ensure confidentiality and authorization. We already described how clients achieve data integrity by verifying the contents of files/chunks by their tokens.

To prevent adversaries from passively reading or actively modifying content while in transmission, the client and proxy first derive a symmetric encryption key K_E before transmitting a chunk. As the token T_{F_i} already serves

[2] Our implementation is structured using asynchronous events and callbacks within a single process, we use the term "thread"here only for explanatory clarity.

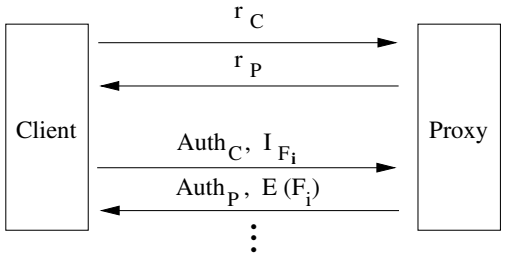


Figure 3: Shark session establishment protocol

as a shared secret for chunk F_i, the parties can simply use it to generate this key.

Figure 3 shows the protocol by which Shark clients establish a secure session. First, the parties exchange fresh, random 20-byte nonces r_C and r_P upon initiating a connection. For each chunk to be sent over the connection, the client must signal the proxy which token T_{F_i} to use, but it can do so without exposing information to eavesdroppers or malicious proxies by simply sending I_{F_i} in the clear. Using these nonces and knowledge of T_{F_i}, each party computes authentication tokens as follows:

$$\begin{aligned} Auth_C &= \mathsf{HMAC}_{T_{F_i}}(\mathsf{A}_C, C, P, r_C, r_P) \\ Auth_P &= \mathsf{HMAC}_{T_{F_i}}(\mathsf{A}_P, P, C, r_P, r_C) \end{aligned}$$

The $Auth_C$ token proves to the proxy that the client actually has the corresponding chunk token T_{F_i} and thus read permissions on the chunk. Upon verifying $Auth_C$, the proxy replies with $Auth_P$ and the chunk F_i after applying E to it.

In our current implementation, E is instantiated by a symmetric block encryption function, followed by an MAC covering the ciphertext. However, we note that $Auth_P$ already serves as a MAC for the *content*, and thus this additional MAC is not strictly needed. [3] The symmetric encryption key K_E for E is derived in a similar manner as before:

$$K_E = \mathsf{HMAC}_{T_{F_i}}(\mathsf{E}, C, P, r_C, r_P)$$

An additional MAC key can be similarly derived by replacing the special character E with M. Shark's use of fresh nonces ensure that these derived authentication tokens and keys cannot be replayed for subsequent requests.

Upon deriving this symmetric key K_E, the proxy encrypts the data within a chunk using 128-bit AES in counter mode (AES-CTR). Per each 16-byte AES block, we use the block's offset within the chunk/file as its counter.

The proxy protocol has READ and READDIR RPCs similar to NFS, except they specify the indexing key I and $Auth_C$ to name a file (which is server independent), in place of a file handle. Thus, after establishing a connection, the client begins issuing read RPCs to the proxy; the client decrypts any data it receives in response using K_E and the proper counter (offset).

While this block encryption prevents a client without T_{F_i} from decrypting the data, one may be concerned if some unauthorized client can download a large number of encrypted blocks, in the hopes of either learning K_E later or performing some offline attack. The proxy's explicit check of $Auth_C$ prevents this. Similarly, the verifiable $Auth_P$ prevents a malicious party that does not hold F_i from registering itself under the public I_{F_i} and then wasting the client's bandwidth by sending invalid blocks (that later will fail hash verification).

Thus, Shark provides strong data integrity guarantees to the client and authorization guarantees to the proxy, even in the face of malicious participants.

2.4 Exploiting file commonalities

We describe the chunking method by which Shark can leverage file commonalities. This method (used by LBFS [22]) avoids a sensitivity to file-length changes by setting chunk boundaries, or *breakpoints*, based on file contents, rather than on offset position. If breakpoints were selected only by offset—for instance, by breaking a file into aligned 16KB chunks—a single byte added to the front of a file would change all breakpoints and thus all chunk tokens.

To divide a file into chunks, we examine every overlapping 48-byte region, and if the low-order 14 bits of the region's Rabin fingerprint [25] equals some globally-chosen value, the region constitutes a breakpoint. Assuming random data, the expected chunk size is therefore $2^{14} = 16$KB. To prevent pathological cases (such as long strings of 0), the algorithm uses a minimum chunk size of 2KB and maximum size of 64KB. Therefore, modifications within a chunk will minimize changes to the breakpoints: either only the chunk will change, one chunk will split into two, or two chunks will merge into one.

Content-based chunking enables Shark to exploit file commonalities: Even if proxies were reading different versions of the same file or different files altogether, a client can discover and download common data chunks, as long as they share the same chunk token (and no server-specific randomness). As the fingerprint value is global, this chunking commonality also persists across multiple file systems.

[3] The results of Krawczyk [15] speaking on the *generic* security concerns of "authentication-and-encrypt" are not really relevant here, as we already expose the raw output of our MAC via I_{F_i} and thus implicitly assume that HMAC does not leak any information about its contents. Thus, the inclusion of $Auth_P$ does not introduce any *additional* data confidentiality concerns.

2.5 Distributed indexing

Shark seeks to enable data sharing both between files on the same file system that contain identical data chunks across different file systems. This functionality is not supported by the simple server-based approach of indexing clients, whereby the file server stores and returns information on which clients are caching which chunks. Thus, we use a *global* distributed index for all Shark clients, even those accessing different Shark file systems.

Shark uses a structured routing overlay [33, 26, 29, 37, 19] to build its distributed index. The system maps opaque keys onto nodes by hashing their value onto a semantic-free identifier (ID) space; nodes are assigned identifiers in the same ID space. It allows scalable key lookup (in $O(\log(n))$ overlay hops for n-node systems), reorganizes itself upon network membership changes, and provides robust behavior against failure.

While many routing overlays optimize routes along the underlay, most are designed as part of distributed hash tables to store immutable data. In contrast, Shark stores only small references about which clients are caching what data: It seeks to allow clients to locate *copies* of data, not merely to find network efficient *routes* through the overlay. In order to achieve such functionality, Shark uses Coral [10] as its distributed index.

System overview. Coral exposes two main protocols: put and get. A Shark client executes the get protocol with its indexing key I as input; the protocol returns a list of proxy addresses that corresponds to some *subset* of the unexpired addresses put under I, taking locality into consideration. put takes as input I, a proxy's address, and some expiry time.

Coral provides a *distributed sloppy hash table* (DSHT) abstraction, which offers weaker consistency than traditional DHTs. It is designed for soft-state where multiple values may be stored under the same key. This consistency is well-suited for Shark: A client need not discover *all* proxies for a particular file, it only needs to find *several*, *nearby* proxies.

Coral caches key/value pairs at nodes whose IDs are close (in terms of identifier space distance) to the key being referenced. To lookup the client addresses associated with a key I, a node simply traverses the ID space with RPCs and, as soon as it finds a remote peer storing I, it returns the corresponding list of values. To insert a key/value pair, Coral performs a two-phase operation. In the "forward" phase, Coral routes to nodes successively closer to I and stops when happening upon a node that is both *full* (meaning it has reached the maximum number of values for the key) and *loaded* (which occurs when there is heavy write traffic for a particular key). During the "reverse" phase, the client node attempts to insert the value at the closest node seen. See [10] for more details.

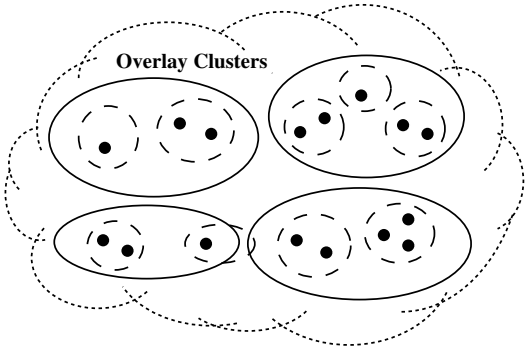


Figure 4: *Coral's three-level hierarchical overlay structure.* Nodes (solid circles) initially query others in their same high-level clusters (dashed rings), whose pointers reference other proxies caching the data within the same small-diameter cluster. If a node finds such a mapping to a replica proxy in the highest-level cluster, the get finishes. Otherwise, it continues among farther, lower-level nodes (solid rings), and finally, if need be, to any node within the system (the cloud).

To improve locality, these routing operations are not initially performed across the entire global overlay: Each Coral node belongs to several distinct routing structures called *clusters*. Each cluster is characterized by a maximum desired network round-trip-time (RTT) called the *diameter*. The system is parameterized by a fixed hierarchy of diameters, or *levels*. Every node belongs to one cluster at each level, as shown in Figure 4. Coral queries nodes in fast clusters before those in slower clusters. This both reduces the latency of lookups and increases the chances of returning values stored by nearby nodes.

Handle concurrency via "atomic" put/get. Ideally, Shark clients should fetch each file chunk from a Shark server only once. However, a DHT-like interface which exposes two methods, put and get, is not sufficient to achieve this behavior. For example, if clients were to wait until completely fetching a file before referencing themselves, other clients simultaneously downloading the file will start transferring file contents from the server. Shark mitigates this problem by using Coral to request *chunks*, as opposed to whole files: A client delays its announcement for only the time needed to fetch a chunk.

Still, given that Shark is designed for environments that may experience abrupt flash crowds—such as when testbed or grid researchers fire off experiments on hundreds of nodes almost simultaneously and reference large executables or data files when doing so—we investigated the practice of clients optimistically inserting a mapping to themselves upon initiating a request. A production use of

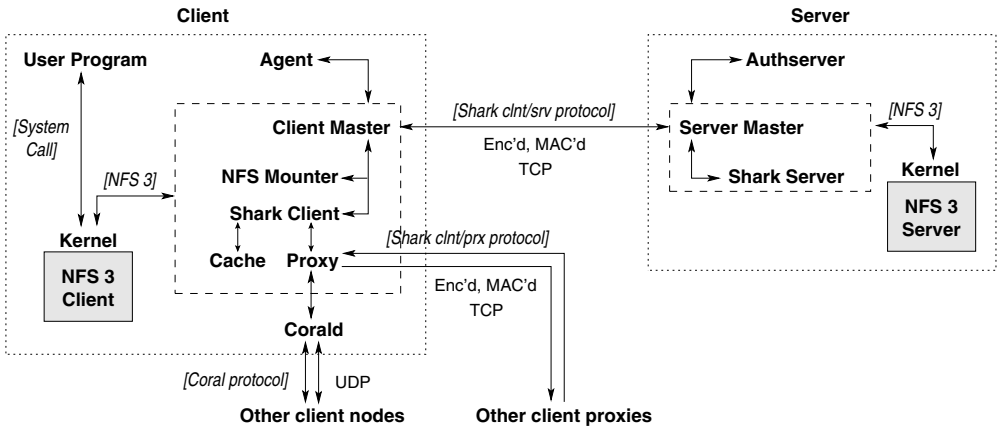


Figure 5: The Shark system components

Coral in a web-content distribution network takes a similar approach when fetching whole web objects [10].

Even using this approach, we found that an origin server can see redundant downloads of the same file when initial requests for a newly-popular file occur synchronously. We can imagine this condition occurring in Shark when users attempt to simultaneously install software on all test-bed hosts.

Such redundant fetches occur under the following race condition: Consider that a mapping for file F (and thus I_F) is not yet inserted into the system. Two nodes both execute $get(I_F)$, then perform a put. On the node closest to I_F, the operations serialize with both gets being handling (and thus returning no values) before either put.

Simply inverting the order of operations is even worse. If multiple nodes first perform a put, followed by a get, they can discover one another and effectively form cycles waiting for one another, with nobody actually fetching the file from the server.

To eliminate this condition, we extended store operations in Coral to provide return status information (like test-and-set in shared-memory systems). Specifically, we introduce a single put/get RPC which atomically performs both operations. The RPC behaves similar to a put as described above, but also returns the first values discovered in *either* direction. (Values in the forward put direction help performance; values in the reverse direction prevent this race condition.)

While of ultimately limited use in Shark given small chunk sizes, this extension also proved beneficial for other applications seeking a distributed index abstraction [10].

3 Implementation

Shark consists of three main components, the server-side daemon `sharksd`, the client-side daemon `sharkcd` and the coral daemon `corald`, as shown in Figure 5. All three components are implemented in C++ and are built using the SFS toolkit [20]. The file-system daemons interoperate with the SFS framework, using its automounter, authentication daemon, etc. `corald` acts as a node within the Coral indexing overlay; a full description can be found in [10].

`sharksd`, the server-side daemon, is implemented as a loop-back client which communicates with the kernel NFS server. `sharksd` incorporates an extension of the NFSv3 protocol—the GETTOK RPC—to support file- and chunk-token retrieval. When `sharksd` receives a GETTOK call, it issues a series of READ calls to the kernel NFS server and computes the tokens and chunk breakpoints. It caches these tokens for future reference. `sharksd` required an additional 400 lines of code to the SFS read-write server.

`sharkcd`, the client-side daemon, forms the biggest component of Shark. In addition to handling user requests, it transparently incorporates whole-file caching and the client- and server-side functionality of the Shark cooperative cache. The code is 12,000 lines.

`sharkcd` comprises an NFS loop-back server which traps user requests and forwards them to either the origin file server or a Shark proxy. In particular, a read for a file block is intercepted by the loop-back server and translated into a series of READ calls to fetch the entire file. The cache-management subsystem of `sharkcd` stores all files that are being fetched locally on disk. This cache provides a thin wrapper around file-system calls to enforce disk usage accounting. Currently, we use the LRU mechanism to evict files from the cache. The cache names are also chosen carefully to fit in the kernel name cache.

The server side of the Shark cooperative cache implements the proxy, accepting connections from other clients. If this proxy cannot immediately satisfy a request, it registers a callback for the request, responding when the block

has been fetched. The client side of the Shark cooperative cache implements the various fetching mechanism discussed in Section 2.3. For every file to be fetched, the client maintains a vector of objects representing connections to different proxies. Each object is responsible for fetching a sequence of chunks from the proxy (or a range of blocks when chunking is not being performed and nodes query only by file token).

`sharkcd` also supports the use of **xfs**, a device driver bundled with the ARLA [35] implementation of AFS, instead of NFS. However, given that the PlanetLab environment on which we performed testing does not have **xfs**, we do not present those results in this paper.

During Shark's implementation, we discovered and fixed several bugs in both the OpenBSD NFS server and the **xfs** implementation.

4 Evaluation

This section evaluates Shark against NFSv3 and SFS to quantify the benefits of its cooperative-caching design for read-heavy workloads. To measure the performance of Shark against these file systems, without the gain from cooperative caching, we first present microbenchmarks for various types of file-system access tests, both in the local-area and across the wide-area. We also evaluate the efficacy of Shark's chunking mechanism in reducing redundant transfers.

Second, we measure Shark's cooperative caching mechanism by performing read tests both within the controlled Emulab LAN environment [36] and in the wide-area on the PlanetLab v3.0 test-bed [24]. In all experiments, we start with cold file caches on all clients, but first warm the server's chunk token cache. The server required 0.9 seconds to compute chunks for a 10 MB random file, and 3.6 seconds for a 40 MB random file.

We chose to evaluate Shark on Emulab, in addition to wide-area tests on PlanetLab, in order to test Shark in a more controlled, native environment: While Emulab allows one to completely reserve machines, individual PlanetLab hosts may be executing tens or hundreds of experiments (slices) simultaneously. In addition, most PlanetLab hosts implement bandwidth caps of 10 Mb/sec across all slices. For example, on a local PlanetLab machine operating at NYU, a Shark client took approximately 65 seconds to read a 40 MB file from the local (non-PlanetLab) Shark file server, while a non-PlanetLab client on the same network took 19.3 seconds. Furthermore, deployments of Shark on large LAN clusters (for example, as part of grid computing environments) may experience similar results to those we report.

The server in all the microbenchmarks and the PlanetLab experiments is a 1.40 GHz Athlon at NYU, running OpenBSD 3.6 with 512 MB of memory. It runs the corresponding server daemons for SFS and Shark. All microbenchmark and PlanetLab clients used in the experiments ran Fedora Core 2 Linux. The server used for Emulab tests was a host in the Emulab test-bed; it did not simultaneously run a client. All Emulab hosts ran RedHat Linux 9.0. Both SFS and Shark issued READ RPCs over TCP for blocks of 8 KB (the packet MTU on FC2's loopback interface is limited to 16 KB by default; we were unable to modify this default for our PlanetLab experiments). NFS, on the other hand, issued read requests over UDP for blocks of 32 KB, requiring four times fewer RPCs and thus significantly less overhead.

4.1 Alternate cooperative protocols

This section considers several alternative cooperative-caching strategies for Shark in order to characterize the benefits of various design decisions.

First, we examine whether clients should issue requests for chunks sequentially (seq), as opposed to choosing a $random$ (previously unread) chunk to fetch. There are two additional strategies to consider when performing sequential requests: Either the client immediately pre-announces itself for a particular chunk upon requesting it (per an "atomic" put/get as in Section 2.5), or the client waits until it finishes fetching a chunk before announcing itself (via a put).

Second, we disable the $negotiation$ process by which clients may reuse connections with proxies and thus download multiple chunks once connected. In this case, the client must query the distributed index for each chunk. We consider such sequential strategies to examine the effect of disk scheduling latency: for single clients in the local area, one intuits that the random strategy limits the throughput to that imposed by the file server's disk seek time, while we expect the network to be the bottleneck in the wide area. Yet, one intuits that when multiple clients operate concurrently, the random strategy allows all clients to fetch independent chunks from the server and later trade these chunks among themselves. Using a purely sequential strategy, the clients all advance only as fast as the few clients that initially fetch chunks from the server.

4.2 Microbenchmarks

For the local-area microbenchmarks, we used a local machine at NYU as a Shark client. Maximum TCP throughput between the local client and server, as measured by `ttcp`, was 11.14 MB/sec. For wide-area microbenchmarks, we used a client machine located at the University of Texas at El Paso. The average round-trip-time (RTT) between this host and the server, as measured by `ping`, is 67 ms. Maximum TCP throughput was 1.07 MB/sec.

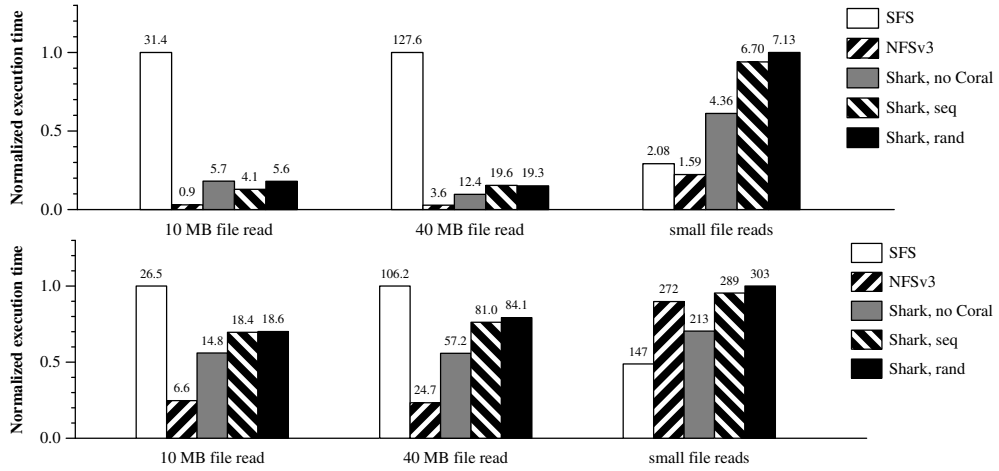


Figure 6: *Local-area* (top) *and wide-area* (bottom) *microbenchmarks.* Normalized application performance for various types of file-system access. Execution times in seconds appear above the bars.

Access latency. We measure the time necessary to perform four types of file-system accesses: (1) to read 10 MB and (2) 40 MB large random files on remote hosts, and (3) to read large numbers of small files. The small file test attempts to read 1,000 1 KB files evenly distributed over ten directories.

We performed single-client microbenchmarks to measure the performance of Shark. Figure 6 shows the performance on the local- and wide-area networks for these three experiments, We compare SFS, NFS, and three Shark configurations, *viz* Shark without calls to its distributed indexing layer ($nocoral$), fetching chunks from a file sequentially (seq), and fetching chunks in random order ($rand$). Shark issues up to eight outstanding RPCs (for seq and $rand$, fetching four chunks simultaneously with two outstanding RPCs per chunk). SFS sends RPCs as requested by the NFS client in the kernel.

For all experiments, we report the normalized *median* value over three runs. We interleaved the execution of each of the five file systems over each run. We see that Shark is competitive across different file system access patterns and is optimized for large read operations.

Chunking. In this microbenchmark, we validate that Shark's chunking mechanism reduces redundant data transfers by exploiting data commonalities.

We first read the `tar` file of the entire source tree for emacs v20.6 over a Shark file system, and then read the `tar` file of the entire source tree for emacs v20.7. We note that of the 2,083 files or directories that comprise these two file archives, 1,425 have not changed between versions (*i.e.*, they have the identical `md5` sum), while 658 of these have changed.

Figure 7 shows the amount of bandwidth savings that the chunking mechanism provides when reading the newer emacs version. When `emacs-20.6.tar` has been cached, Shark only transfers 33.8 MB (1416 chunks) when reading `emacs-20.7.tar` (of size 56.3 MB).

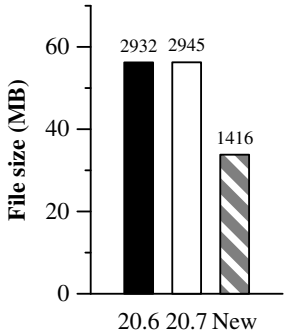


Figure 7: *Bandwidth savings from chunking.* "New" reflects the number of megabytes that need to be transferred when reading emacs 20.7 given 20.6. Number of chunks comprising each transfer appears above the bars.

4.3 Local-area cooperative caching

Shark's main claim is that it improves a file server's scalability, which retaining its benefits. We now study the end-to-end performance of reads in a cooperative environment with many clients attempting to simultaneously read the same file(s).

In this section, we evaluate Shark on Emulab [36]. These experiments allowed us to evaluate various cooperative strategies in a better controlled environment. In all the configurations of Shark, clients attempt to download a file from four other proxies simultaneously.

Figure 8 shows the cumulative distribution functions (CDFs) of the time needed to read a 10 MB and 40 MB (random) file across 100 physical Emulab hosts,

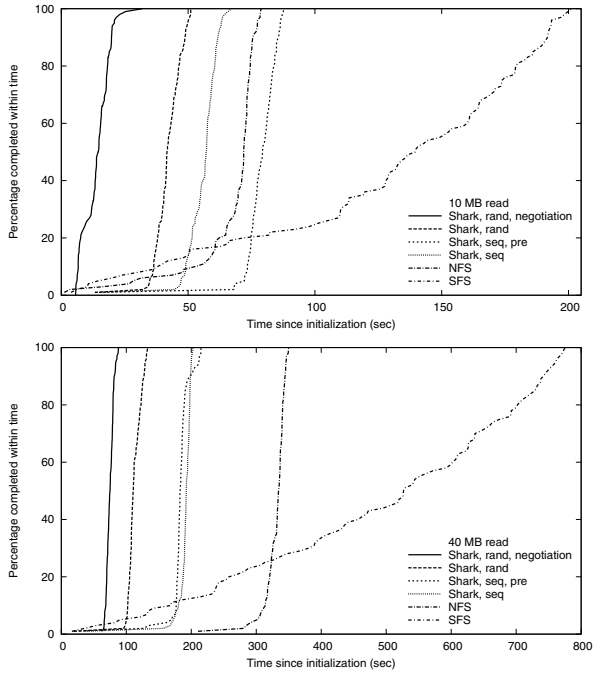


Figure 8: *Client latency.* Time (seconds) for ˜100 LAN hosts to read a 10 MB (top) and 40 MB (bottom) file.

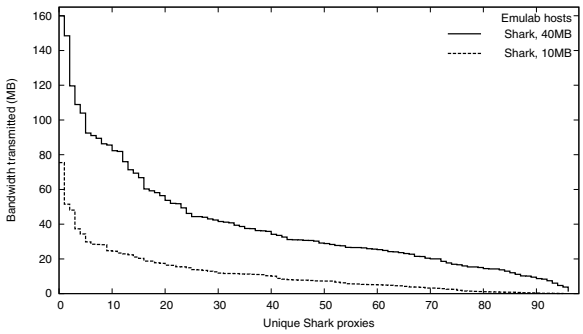


Figure 9: *Proxy bandwidth usage.* MBs served by each Emulab proxy when reading 40 MB and 10 MB files.

comparing various cooperative read strategies of Shark, against vanilla SFS and NFS. In each experiment, all hosts mounted the server and began fetching the file simultaneously. We see that Shark achieves a median completion time $< \frac{1}{4}$ that of NFS and $< \frac{1}{6}$ that of SFS. Furthermore, its 95th percentile is almost an order of magnitude better than SFS.

Shark's fast, almost vertical rise (for nearly all strategies) demonstrates its cooperative cut-through routing: Shark clients effectively organize themselves into a distribution mesh. Considering a single data segment, a client is part of a chain of nodes performing cut-through routing, rooted at the origin server. Because clients may act as root nodes for some blocks and act as leaves for others, most finish at almost synchronized times. The lack of any degradation of performance in the upper percentiles demonstrates the lack of any heterogeneity, in terms of both network bandwidth and underlying disk/CPU load, among the Emulab hosts.

Interestingly, we see that most NFS clients finish at loosely synchronized times, while the CDF of SFS clients' times has a much more gradual slope, even though both systems send all read requests to the file server. Subsequent analysis of NFS over TCP (instead of NFS over UDP as shown) showed a similar slope as SFS, as did Shark without its cooperative cache. One possible explanation is that the heavy load on (and hence congestion at) the file server imposed by these non-cooperative file systems causes TCP to back-off, greatly reducing system throughput.

We find that a $random$ request strategy, coupled with inter-proxy $negotiation$, distinctly outperforms all other evaluated strategies. A sequential strategy effectively saw the clients furthest along in reading a file fetch the leading (four) chunks from the origin file server; other clients used these leading clients as proxies. Thus, modulo possible inter-proxy timeouts and synchronous requests in the non-pre-announce example, the origin server saw at most four simultaneous chunk requests. Using a $random$ strategy, more chunks are fetched from the server simultaneously and thus propagate quicker through the clients' dissemination mesh.

Figure 9 shows the total amount of bandwidth served by each proxy as part of Shark's cooperative caching, when using a random fetch strategy with inter-proxy negotiation for the 40 MB and 10 MB experiments. We see that the proxy serving the most bandwidth contributed four and seven times more upstream bandwidth than downstream bandwidth, respectively. During these experiments, the Shark file server served a total of 92.55 MB and 15.48 MB, respectively. Thus, we conclude that Shark is able to significantly reduce bandwidth a file server's bandwidth utilization, even when distributing files to large numbers of clients. Furthermore, Shark ensures that any one cooperative-caching client does not assume excessive bandwidth costs.

4.4 Wide-area cooperative caching

Shark's main claim is that it improves a file server's scalability, which still maintaining security, accountability, etc. In our cooperative caching experiment, we study the end-to-end performance of attempting to perform reads within a large, wide-area distributed test-bed.

On approximately 185 PlanetLab hosts, well-distributed from North America, Europe, and Asia, we attempted to simultaneously read a 40 MB random

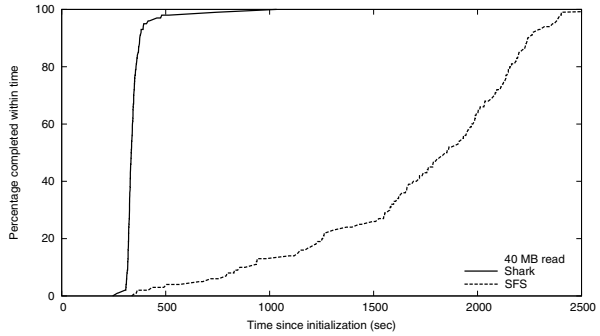


Figure 10: *Client latency.* Time (seconds) for 185 hosts to finish reading a 40 MB file using Shark and SFS.

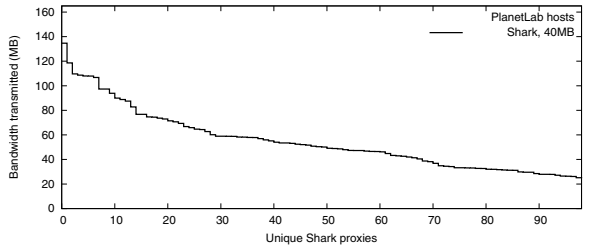


Figure 11: *Proxy bandwidth usage.* MBs served by each PlanetLab proxy when reading 40 MB files.

file. All hosts mounted the server and began fetching the file simultaneously.

Figure 10 shows a CDF of the time needed to read the file on all hosts, comparing Shark with SFS.

% done in (sec)	50%	75%	90%	95%	98%
Shark	334	350	375	394	481
SFS	1848	2129	2241	2364	2396

We see that, between the 50th and 98th percentiles, Shark is five to six times faster than SFS. The graph's sharp rise and distinct knee demonstrates Shark's cooperative caching: 96% of the nodes effectively finish at nearly the same time. Clients in SFS, on the other hand, complete at a much slower rate.

Wide-area experiments with NFS repeatedly crashed our file server (*i.e.*, it caused a kernel panic). We were therefore unable to evaluate NFS in the wide area.

Figure 10 shows the total amount of bandwidth served by each proxy during this experiment. We see that the proxy serving the most bandwidth contributed roughly three times more upstream than downstream bandwidth.

Figure 12 shows the number of bytes read from our file server during the execution of these two experiments. We see that Shark reduces the server's bandwidth usage by an order of magnitude. In fact, we believe that the Shark's client cache implementation can be improved to reduce bandwidth usage quite further: We are currently examining the trade-offs between continually retrying the cooperative cache and increased client latency.

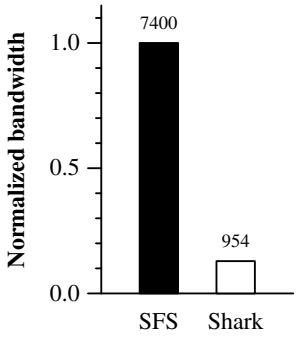


Figure 12: *Server bandwidth usage.* Megabytes read from server as a 40 MB file is fetched by 185 hosts.

5 Related Work

There are numerous network file systems designed for local-area access. NFS [31] provides a server-based file system, while AFS [14] improves its performance via client-side caching. Some network file systems provide security to operate on untrusted networks, including AFS with Kerberos [32], Echo [18], Truffles [27], and SFS [21]. Even wide-area file systems such as AFS do not perform any bandwidth optimizations necessary for types of workloads and applications Shark targets. Additionally, although not an intrinsic limitation of AFS, there are some network environments that do not work as well with its UDP-based transport compared to a TCP-based one. This section describes some complementary and alternate designs for building scalable file systems.

Scalable file servers. JetFile [12] is a wide-area network file system designed to scale to large numbers of clients, by using the Scalable Reliable Multicast (SRM) protocol, which is logically layered on IP multicast. JetFile allocates a multicast address for each file. Read requests are multicast to this address; any client which has the data responds to such requests. In JetFile, any client can become the manager for a file by writing to it—which implies the necessity for conflict-resolution mechanisms to periodically synchronize to a storage server—whereas all writes in Shark are synchronized at a central server. However, this practice implies that JetFile is intended for read-write workloads, while Shark is designed for read-heavy workloads.

High-availability file systems. Several local-area systems propose distributing functionality over multiple collocated hosts to achieve greater fault-tolerance and availability. Zebra [13] uses a single meta-data server to serialize meta-data operations (*e.g.* i-node operations), and maintains a per-client log of file contents striped across multiple network nodes. Harp [17] replicates file servers to ensure high availability; one such server acts as a pri-

mary replica in order to serialize updates. These techniques are largely orthogonal to, yet possibly could be combined with, Shark's cooperative caching design.

Serverless file systems. Serverless file systems are designed to offer greater local-area scalability by replicating functionality across multiple hosts. xFS [3] distributes data and meta-data across all participating hosts, where every piece of meta-data is assigned a host at which to serialize updates for that meta-data. Frangipani [34] decentralizes file-storage among a set virtualized disks, and it maintains traditional file system structures, with small meta-data logs to improve recoverability. A Shark server can similarly use any type of log-based or journaled file system to enable recoverability, while it is explicitly designed for wide-area scalability.

Farsite [1] seeks to build an enterprise-scale distributed file system. A single primary replica manages file writes, and the system protects directory meta-data through a Byzantine-fault-tolerant protocol [7]. When enabling cross-file-system sharing, Shark's encryption technique is similar to Farsite's convergent encryption, in which files with identical content result in identical ciphertexts.

Peer-to-peer file systems. A number of peer-to-peer file systems—including PAST [30], CFS [8], Ivy [23], and OceanStore [16]—have been proposed for wide-area operation and similarly use some type of distributed-hash-table infrastructure ([29, 33, 37], respectively). All of these systems differ from Shark in that they provide a serverless design: While such a decentralized design removes any central point of failure, it adds complexity, performance overhead, and management difficulties.

PAST and CFS are both designed for read-only data, where data (whole files in PAST and file blocks in CFS) are *stored* in the peer-to-peer DHT [29, 33] at nodes closest to the key that names the respective block/file. Data replication helps improve performance and ensures that a single node is not overloaded. In contract, Shark uses Coral to *index* clients caching a replica, so data is only cached where it is needed by applications and on nodes who have proper access permissions to the data.

Ivy builds on CFS to yield a read-write file system through logs and version vectors. The head of a per-client log is stored in the DHT at its closest node. To enable multiple writers, Ivy uses version vectors to order records from different logs. It does not guarantee read/write consistency. Also managing read/write storage via versioned logs, OceanStore divides the system into a large set of untrusted clients and a core group of trusted servers, where updates are applied atomically. Its Pond prototype [28] uses a combination of Byzantine-fault-tolerant protocols, proactive threshold signatures, erasure-encoded and block replication, and multicast dissemination.

Large file distribution. BitTorrent [5] is a widely-deployed file-distribution system. It uses a central server to track which clients are caching which blocks; using information from this meta-data server, clients download file blocks from other clients in parallel. Clients access BitTorrent through a web interface or special software.

Compared to BitTorrent, Shark provides a file-system interface supporting read/write operations with flexible access control policies, while BitTorrent lacks authorization mechanisms and supports read-only data. While BitTorrent centralizes client meta-data information, Shark stores such information in a global distributed index, enabling cross-file-system sharing (for world-readable files) and taking advantage of network locality.

6 Conclusion

We argue for the utility of a network file system that can scale to thousands of clients, while simultaneously providing a drop-in replacement for local-area file systems. We present Shark, a file system that exports existing local file systems, ensures compatibility with existing administrative procedures, and provides performance competitive with other secure network file systems on local-area networks. For improved wide-area performance, Shark clients construct a locality-optimized cooperative cache by forming self-organizing clusters of well-connected machines. They efficiently locate nearby copies of data using a distributed index and stripe downloads from multiple proxies. This simultaneously reduces the load on file servers and delivers significant performance improvements for the clients. In doing so, Shark appears promising for achieving the goal of a scalable, efficient, secure, and easily-administered distributed file system.

Acknowledgments. We thank Vijay Karamcheti, Jinyuan Li, Robert Grimm, our shepherd, Peter Druschel, and members of NYU systems group for their helpful feedback on drafts of this paper. We would like to thank Emulab (Robert Ricci, Timothy Stack, Leigh Stoller, and Jay Lepreau) and PlanetLab (Steve Muir and Larry Peterson) researchers for assistance in running file-system experiments on their test-beds, as well as Eric Freudenthal and Jayanth Kumar Kannan for remote machine access. Finally, thanks to Jane-Ellen Long at USENIX for her consideration.

This research was conducted as part of the IRIS project (`http://project-iris.net/`), supported by the NSF under Cooperative Agreement No. ANI-0225660. Michael Freedman is supported by an NDSEG Fellowship. David Mazières is supported by an Alfred P. Sloan Research Fellowship.

References

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI*, Boston, MA, December 2002.

[2] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient overlay networks. In *SOSP*, pages 131–145, Banff, Canada, October 2001.

[3] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roseli, and R. Y. Wang. Serverless network file systems. *ACM Trans. on Computer Systems*, 14(1):41–79, February 1996.

[4] M. Bellare, R. Canetti, and H. Krawczyk. Keyed hash functions and message authentication. In *Advances in Cryptology—CRYPTO '96*, Santa Barbara, CA, August 1996.

[5] BitTorrent. http://www.bittorrent.com/, 2005.

[6] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification. RFC 1813, Network Working Group, June 1995.

[7] M. Castro and B. Liskov. Proactive recovery in a byzantine-fault-tolerant system. In *OSDI*, San Diego, October 2000.

[8] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *SOSP*, Banff, Canada, Oct 2001.

[9] D. H. J Epema, Miron Livny, R. van Dantzig, X. Evers, and Jim Pruyne. A worldwide flock of condors: Load sharing among workstation clusters. *J. Future Generations of Computer Systems*, 12:53–65, 1996.

[10] M. J. Freedman, E. Freudenthal, and D. Maziéres. Democratizing content publication with Coral. In *NSDI*, San Francisco, CA, March 2004.

[11] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP*, pages 202–210, December 1989.

[12] B. Gronvall, A. Westerlund, and S. Pink. The design of a multicast-based distributed file system.

[13] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. In *SOSP*, December 1993.

[14] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. on Computer Systems*, 6(1):51–81, February 1988.

[15] H. Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is ssl?). In *Advances in Cryptology—CRYPTO 2001*, Santa Barbara, CA, 2001.

[16] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *ASPLOS*, Cambridge, MA, Nov 2000.

[17] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. eplication in the Harp file system. *Operating Systems Review*, 25(5):226–238, October 1991.

[18] T. Mann, A. D. Birrell, A. Hisgen, C. Jerian, and G. Swart. A coherent distributed file cache with directory write-behind. *ACM Trans. on Computer Systems*, 12(2):123–164, May 1994.

[19] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS*, Cambridge, MA, Mar 2002.

[20] D. Mazières. A toolkit for user-level file systems. In *USENIX*, Boston, MA, Jun 2001.

[21] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *SOSP*, Kiawah Island, SC, December 1999.

[22] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *SOSP*, October 2001.

[23] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *OSDI*, Boston, MA, December 2002.

[24] PlanetLab. http://www.planet-lab.org/, 2005.

[25] M. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[26] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *ACM SIGCOMM*, San Diego, CA, August 2001.

[27] P. Reiher, Jr. T. Page, G. J. Popek, J. Cook, and S. Crocker. Truffles—a secure service for widespread file sharing. In *PSRG Workshop on Network and Distributed System Security*, San Diego, CA, 1993.

[28] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *FAST*, Berkeley, CA, March 2003.

[29] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware*, November 2001.

[30] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *SOSP*, Banff, Canada, October 2001.

[31] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Summer 1985 USENIX*, Portland, OR, June 1985.

[32] J. G. Steiner, B. C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Winter 1988 USENIX*, Dallas, TX, February 1988.

[33] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. In *IEEE/ACM Trans. on Networking*, 2002.

[34] C. Thekkath, T. Mann, and E Lee. Frangipani: A scalable distributed file system. In *SOSP*, Saint Malo, France, October 1997.

[35] A. Westerlund and J. Danielsson. Arla—afree AFS client. In *1998 USENIX, Freenix track*, New Orleans, LA, June 1998.

[36] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI*, Boston, MA, December 2002.

[37] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE J. Selected Areas in Communications*, 22(1):41–53, 2003.

Glacier: Highly durable, decentralized storage despite massive correlated failures

Andreas Haeberlen Alan Mislove Peter Druschel

Department of Computer Science, Rice University
{ahae,amislove,druschel}@cs.rice.edu

Abstract

Decentralized storage systems aggregate the available disk space of participating computers to provide a large storage facility. These systems rely on data redundancy to ensure durable storage despite of node failures. However, existing systems either assume independent node failures, or they rely on introspection *to carefully place redundant data on nodes with low expected failure correlation. Unfortunately, node failures are not independent in practice and constructing an accurate failure model is difficult in large-scale systems. At the same time, malicious worms that propagate through the Internet pose a real threat of large-scale correlated failures. Such rare but potentially catastrophic failures must be considered when attempting to provide highly durable storage.*

In this paper, we describe Glacier, a distributed storage system that relies on massive redundancy to mask the effect of large-scale correlated failures. Glacier is designed to aggressively minimize the cost of this redundancy in space and time: Erasure coding and garbage collection reduces the storage cost; aggregation of small objects and a loosely coupled maintenance protocol for redundant fragments minimizes the messaging cost. In one configuration, for instance, our system can provide six-nines durable storage despite correlated failures of up to 60% of the storage nodes, at the cost of an elevenfold storage overhead and an average messaging overhead of only 4 messages per node and minute during normal operation. Glacier is used as the storage layer for an experimental serverless email system.

1 Introduction

Distributed, cooperative storage systems like FarSite and OceanStore aggregate the often underutilized disk space and network bandwidth of existing desktop computers, thereby harnessing a potentially huge and self-scaling storage resource [1, 27]. Distributed storage is also a fundamental component of many other recent decentralized systems, for instance, cooperative backup, serverless messaging or distributed hash tables [15, 17, 20, 28, 31].

Since individual desktop computers are not sufficiently dependable, redundant storage is typically used in these systems to enhance data availability. For instance, if nodes are assumed to fail independently with probability p, a system of k replicas fails with probability $p^k \ll p$; the parameter k can be adjusted to achieve the desired level of availability. Unfortunately, the assumption of failure independence is not realistic [3, 4, 25, 41, 43]. In practice, nodes may be located in the same building, share the same network link, or be connected to the same power grid.

Most importantly, many of the nodes may run the same software. Results of our own recent survey of 199 random Gnutella nodes, which is consistent with other statistics [34], showed that 39% of the nodes were using the Morpheus client; more than 80% were running the Windows operating system. A failure or security vulnerability associated with a widely shared software component can affect a majority of nodes within a short period of time. Worse, worms that propagate via email, for instance, can even infect computers within a firewalled corporate intranet.

On the other hand, stored data represents an important asset and has considerable monetary value in many environments. Loss or corruption of business data, personal records, calendars or even user email could have catastrophic effects. Therefore, it is essential that a storage system for such data be sufficiently dependable. One aspect of dependability is the *durability* of a data object, which we define, for the purposes of this paper, as the probability that a specific data object will survive an assumed worst-case system failure.

Large-scale correlated failures can be observed in the Internet, where thousands of nodes are regularly affected by virus or worm attacks. Both the frequency and the severity of these attacks have increased dramatically in recent years [39]. So far, these attacks have rarely caused data losses. However, since the malicious code can often obtain administrator privileges on infected machines, the

attackers could easily have erased the locals disks had they intended to do serious harm.

In this paper, we describe Glacier, a distributed storage system that is robust to large-scale correlated failures. Glacier's goal is to provide highly durable, decentralized storage suitable for important and otherwise unrecoverable data, despite the potential for correlated, Byzantine failures of a majority of the participating storage nodes. Our approach is 'extreme' in the sense that, in contrast to other approaches [23, 27], we assume the exact nature of the correlation to be unpredictable. Hence, Glacier must use redundancy to prepare for a wide range of failure scenarios. In essence, Glacier trades efficiency in storage utilization for durability, thus turning abundance into reliability.

Since Glacier does not make any assumptions about the nature and correlation of faults, it can provide hard, analytical durability guarantees. The system can be configured to prevent data loss even under extreme conditions, such as correlated failures with data loss on 85% of the storage nodes or more. Glacier makes use of erasure codes to spread data widely among the participating storage nodes, thus generating a degree of redundancy that is sufficient to survive failures of this magnitude. Aggregation of small objects and a loosely coupled fragment maintenance protocol reduce the message overhead for maintaining this massive redundancy, while the use of erasure codes and garbage collection of obsolete data mitigate the storage cost.

Despite these measures, there is a substantial storage cost for providing strong durability in such a hostile environment. For instance, to ensure an object survives a correlated failure of 60% of the nodes with a probability of .999999, the storage overhead is about 11-fold. Fortunately, disk space on desktop PCs is a vastly underutilized resource. A recent study showed that on average, as much as 90% of the local disk space is unused [9]. At the same time, disk capacities continue to follow Moore's law [22]. Glacier leverages this abundant but unreliable storage space to provide durable storage for critical data. To the best of our knowledge, Glacier is the first system to provide hard durability guarantees in such a hostile environment.

The rest of this paper is structured as follows: In the next section, we give an overview of existing solutions for ensuring long-term data durability. Section 3 describes the assumptions we made in the design of our system, and the environment it is intended for. In the following two sections, we demonstrate how Glacier can lend a distributed hash table data durability in the face of large-scale correlated failures. We discuss security aspects in Section 6 and describe our experimental evaluation results in Section 7. Finally, Section 8 presents our conclusions.

2 Related work

OceanStore [27] and Phoenix [23, 24] apply *introspection* to defend against the threat of correlated failures. OceanStore relies primarily on inferring correlation by observing actual failures, whereas Phoenix proactively infers possible correlations by looking at the configuration of the system, e.g. their operating system and installed software. In both systems, the information is then used to place replicas of an object on nodes that are expected to fail with low correlation.

However, the failure model can only make accurate predictions if it reflects *all* possible causes of correlated failures. One possible conclusion is that one has to carefully build a very detailed failure model. However, a fundamental limitation of the introspective approach is that observation does not reveal low-incidence failures and it is difficult for humans to predict all sources of correlated failures. For instance, a security vulnerability that exists in two different operating systems due to a historically shared codebase is neither observable, nor are developers or administrators likely to be aware of it prior to its first exploit.

Moreover, introspection itself can make the system vulnerable to a variety of attacks. Selfish node operators may have an incentive to provide incorrect information about their nodes. For example, a user may want to make her node appear less reliable to reduce her share of the storage load, while an attacker may want to do the opposite in an attempt to attract replicas of an object he wants to censor. Finally, making failure-related information available to peers may be of considerable benefit to an attacker, who may use it to choose promising targets.

Introspective systems can achieve robustness to correlated failures at a relatively modest storage overhead, but they assume an accurate failure model, which involves risks that are hard to quantify. Glacier is designed to provide very high data durability for important data. Thus, it chooses a point in the design space that relies on minimal assumptions about the nature of failures, at the expense of larger storage overhead compared to introspective systems.

TotalRecall [5] is an example of a system that uses introspection to optimize availability under churn. Since this system does not give any worst-case guarantees, our criticism of introspection does not apply to it.

OceanStore [27], like Glacier, uses separate mechanisms to maintain short-term availability and to ensure long-term durability. Unlike Glacier, OceanStore cannot sustain Byzantine failures of a large fraction of storage nodes [44].

Many systems use redundancy to guard against data loss. PAST [20] and Farsite [1] replicate objects across multiple nodes, while Intermemory [14], Free-

Haven [18], Myriad [13], PASIS [45] and other systems [2] use erasure codes to reduce the storage overhead for the redundant data. Weatherspoon et al. [42] show that erasure codes can achieve mean time to failures many orders of magnitude higher than replicated systems with similar storage and bandwidth requirements. However, these systems assume only small-scale correlated failures or failure independence. Systems with support for remote writes typically rely on quorum techniques or Byzantine fault tolerance to serialize writes and thus cannot sustain a catastrophic failure.

Cates [12] describes a data management scheme for distributed hashtables that keeps a small number of erasure-coded fragments for each object to decrease fetch latency and to improve robustness against small-scale fail-stop failures. The system is not designed to sustain large-scale correlated failures or Byzantine faults.

Glacier spends a high amount of resources to provide strong worst-case durability guarantees. However, not all systems require this level of protection; in some cases, it may be more cost-effective to optimize for expected failure patterns. Keeton et al. [26] present a quantitative discussion of the tradeoff between cost and dependability.

Glacier uses leases to control the lifetime of stored objects, which need to be periodically renewed to keep an object alive. Leases are a common technique in distributed storage systems; for example, they have been used in Tapestry [46] and CFS [17].

A particularly common example of correlated failures are Internet worm attacks. The course, scope and impact of these attacks has been studied in great detail [29, 30, 38, 39, 47].

3 Assumptions and intended environment

In this section, we describe assumptions that underlie the design of Glacier and the environment it is intended for.

Glacier is a decentralized storage layer providing data durability in the event of large-scale, correlated and Byzantine storage node failures. It is intended to be used in combination with a conventional, decentralized replicating storage layer that handles normal read and write access to the data. This *primary storage* layer might typically keep a small number of replicas of each data object, sufficient to mask individual node failures without loss in performance or short-term availability.

Glacier is primarily intended for an environment consisting of desktop computers within an organizational intranet, though some fraction of nodes are assumed to be notebooks connected via a wireless LAN or home desktops connected via cable modems or DSL. Consistent with this environment, we assume modest amounts of churn and relatively good network connectivity. A substantial fraction of the nodes is assumed to be online most of the time, while the remaining nodes (notebooks and home desktops) may be disconnected for extended periods of time. In the following, we outline key assumptions underlying Glacier's design.

3.1 Lifetime versus session time

We define the *lifetime* of a node as the time from the instant when it first joins the system until it either permanently departs or it loses its locally stored data. The *session time* of a node is the time during which it remains connected to the overlay network. We assume that the expected lifetime of a node is high, at least on the order of several weeks. Without a reasonably long lifetime a cooperative, persistent storage system is infeasible since the bandwidth overhead of moving data between nodes would be prohibitive [6]. Glacier is intended for an environment similar to the one described by Bolosky et al. [8], where an expected lifetime of 290 days was reported.

However, session times can be much shorter, on the order of hours or days. Nodes may go offline and return with their disk contents intact, as would be expected of notebooks, home desktops, or desktops that are turned off at night or during weekends.

3.2 Failure model

We assume that Glacier is in one of three operating modes at any given time: *normal*, *failure* or *recovery*. During *normal operation*, only a small fraction of nodes is assumed to be faulty at any time, though a strong minority of the nodes may be off-line. In this mode, Glacier performs the background tasks of aggregation, coding and storage of newly written data, garbage collection, and fragment maintenance.

During a *large-scale failure*, a majority of the storage nodes, but not more than a fraction f_{max}, have suffered Byzantine failures virtually simultaneously. In this mode, we cannot assume that communication within the system is possible, and Glacier's role is limited to protecting the data stored on non-faulty nodes. It is sufficient to choose f_{max} as a loose upper bound, which can be estimated from the overall amount of diversity in the system. The failure state is assumed to last less than Glacier's *object lease period* L_O.

Glacier enters *recovery mode* when sysadmins have recovered or taken off-line enough of the faulty nodes so that communication within the system is once again possible. In this mode, Glacier reconstitutes aggregates from surviving fragments and restores missing fragments. Note that Glacier does not explicitly differentiate between the three modes.

3.3 Requirements

Glacier assumes that the participating storage nodes form an overlay network. The overlay is expected to provide a distributed directory service that maps numeric keys to the address of a live node that is currently responsible for the key. Glacier assumes that the set of possible keys forms a circular space, where each live participating node is responsible for an approximately uniformly sized segment of the key space. This segment consists of all keys closest to the node's identifier. Participating nodes store objects with keys in their segment. If a node fails, the objects in its local store may be lost.

To prevent Sybil attacks [19], node identifiers are assigned pseudo-randomly and it is assumed that an attacker cannot acquire arbitrarily many legitimate node identifiers. This can be ensured though the use of *certified node identifiers* [10].

Structured overlay networks with a distributed hash table (DHT) layer like DHash/Chord [16, 40] or PAST/Pastry [20, 37] provide such a service, though other implementations are possible. Glacier requires that it can always reliably identify, authenticate and communicate with the node that is currently responsible for a given key. If the overlay provides secure routing techniques, such as those described by Castro et al. [10], then Glacier can tolerate Byzantine failures during normal operation.

Glacier assumes that the participating nodes have loosely synchronized clocks, for instance by running NTP [33]. Glacier does not depend on the correctness of its time source, nor the correctness of the overlay directory services during large-scale failures.

4 Glacier

The architecture of Glacier is depicted in Figure 1. Glacier operates alongside a *primary store*, which maintains a small number of full replicas of each data object (e.g., 2–3 replicas). The primary store ensures efficient read and write access and provides short-term availability of data by masking individual node failures. Glacier acts as an archival storage layer, ensuring long-term durability of data despite large-scale failure. The aggregation layer, described in Section 5, aggregates small objects prior to their insertion into Glacier for efficiency. Objects of sufficient size can be inserted directly into Glacier.

During normal operation, newly written or updated data objects are aggregated asynchronously. Once a sufficiently large aggregate has accumulated or a time limit is reached, Glacier erasure codes the aggregate and places the fragments at pseudo-randomly selected storage nodes throughout the system. Periodically, Glacier consolidates remaining live objects into new aggregates, inserts the new fragments and discards fragments corresponding to old aggregates.

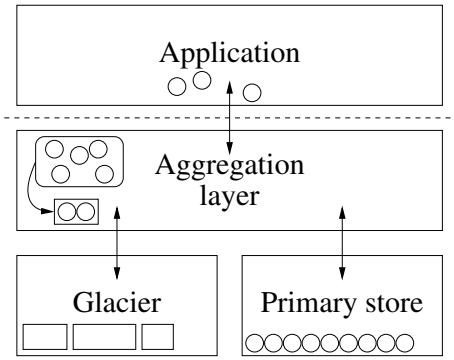


Figure 1. Structure of a multi-tier system with Glacier and an additional aggregation layer.

Once an object is stored as part of an erasure coded aggregate, Glacier ensures that the object can be recovered even if the system suffers from a large-scale, correlated Byzantine failure. The *durability guarantee* given by Glacier implies that, if the failure affects a fraction $f \leq f_{max}$ of the storage nodes, each object survives with probability $P \geq P_{min}$. The parameters f_{max} and P_{min} determine the overhead and can be adjusted to the requirements of the application.

Glacier ensures durability by spreading redundant data for each object over a large number of storage nodes. These nodes periodically communicate with each other to detect data loss, and to re-create redundancy when necessary. After a large-scale failure event, Glacier reconstitutes aggregates from surviving fragments and reinserts objects into the primary store. The recovery proceeds gradually to prevent network overload. Additionally, an on-demand primitive is available to recover objects synchronously when requested by the application.

4.1 Interface to applications

Glacier is designed to protect data against Byzantine failures, including a failures of the node that inserted an object. Therefore, there are no primitives to either delete or overwrite existing data remotely. However, *leases* are used to limit the time for which an object is stored; when its lease expires, the object can be removed and its storage is reclaimed. Application must renew the leases of all objects they care about once per lease period. The lease period is chosen to exceed the assumed maximal duration of a large-scale failure event, typically several weeks or months. Also, since objects in Glacier are ef-

fectively immutable, updated objects must be inserted with a different *version number*.

Applications interact with Glacier by invoking one of the following methods:

- `put(i,v,o,l)` stores an object o under identifier i and version number v, with a lease period of l.
- `get(i,v)`$\rightarrow$`o` retrieves the object stored under identifier i and version number v. If the object is not found, or if its lease has expired, `nil` is returned.
- `refresh(i,v,l)` extends the lease of an existing object. If the current lease period of the object already exceeds l, the operation has no effect.

4.2 Fragments and manifests

Glacier uses an *erasure code* [35] to reduce storage overhead. We use a variant of Reed-Solomon codes based on Cauchy matrices [7], for which efficient codecs exist. However, any other erasure code could be used as well. An object O of size $|O|$ is recoded into n *fragments* $F_1, F_2, \ldots, F_n$ of size $\frac{|O|}{r}$, any r of which contain sufficient information to restore the entire object. If possible, each fragment is stored on a different node, or *fragment holder*, to reduce failure correlation among fragments.

If the object O is stored under a key k, then its fragments are stored under a *fragment key* (k, i, v), where i is the index of the fragment and v is a version number. For each version, Glacier maintains an independent set of fragments. If an application creates new versions frequently, it can choose to bypass Glacier for some versions and apply the corresponding modifications to the primary storage system only.

For each object O, Glacier also maintains an *authenticator*

$$A_O = (H(O), H(F_1), H(F_2), \ldots, H(F_n), v, l)$$

where $H(f)$ denotes a secure hash (e.g., SHA-1) of f. This is necessary to detect and remove corrupted fragments during recovery, since any modification to a fragment would cause the object to be reconstructed incorrectly. The value l represents the lease associated with the object; for permanent objects, the value $l = \infty$ is used.

The authenticator is part of a *manifest* M_O, which accompanies the object and each of its fragments. The manifest may contain a cryptographic signature that authenticates the object and each of its fragments; it can also be used to store metadata such as credentials or billing information. For immutable objects that do not require a specific, chosen key value, it is sufficient to choose $M_O = A_O$ and $k = H(A_O)$; this makes the object and each of its fragments self-certifying.

4.3 Key ownership

In structured overlays like Pastry or Chord, keys are assigned to nodes using consistent hashing. For instance, in Pastry, a key is mapped to the live node with the numerically closest node identifier. In the event of a node departure, keys are immediately reassigned to neighboring nodes in the id space to ensure availability.

In Glacier, this is both unnecessary and undesirable because fragments stored on nodes that are temporarily off-line do not need to be available and therefore do not need to be reassigned. For this reason, Glacier uses a modified assignment of keys to nodes, where keys are assigned by consistent hashing over the set of nodes that are either on-line or were last online within a period T_{max}.

4.4 Fragment placement

In order to determine which node should store a particular fragment (k, i, v), Glacier uses a *placement function* P. This function should have the following properties:

1. Fragments of the same object should be placed on different, pseudo-randomly chosen nodes to reduce inter-fragment failure correlation.
2. It must be possible to locate the fragments after a failure, even if all information except the object's key is lost.
3. Fragments of objects with similar keys should be grouped together so as to allow the aggregation of maintenance traffic.
4. The placement function should be *stable*, i.e. the node on which a fragment is placed should change rarely.

A natural solution would be to use a 'neighbor set', i.e. to map (k, i, v) to the ith closest node relative to k. Unfortunately, this solution is not stable because the arrival of a new node in the vicinity of k would change the placement of most fragments. Also, choosing $P(k, i, v)$ as the content hash of the corresponding fragment is not a solution because it does not allow fragments to be located after a crash. Instead, Glacier uses

$$P(k, i, v) = k + \frac{i}{n+1} + H(v)$$

This function maps the primary replica at position k and its n fragments to $n + 1$ equidistant points in the circular id space (Figure 2). If multiple versions exist, the hash $H(v)$ prevents a load imbalance by placing their fragments on different nodes.

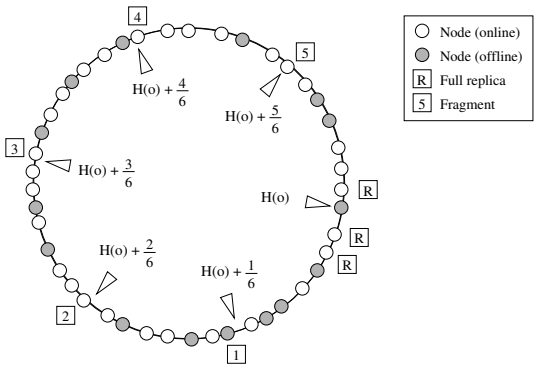


Figure 2. Fragment placement in a configuration with five fragments and three replicas in the primary store.

When a new object (k, v) must be inserted, Glacier uses the overlay to send probe messages to each location $P(k, i, v), i = 1..N$. If the owner of $P(k, i, v)$ is currently online, it responds to this message, and Glacier sends the fragment directly to that node. Otherwise, the fragment is discarded and restored later by the maintenance mechanism.

If the availability of the nodes is very low, there may be situations where fewer than r fragment holders are online during insertion. In this case, the inserting node sends additional probe messages, which are answered by one of the owners' neighbors. These neighbors then act as temporary fragment holders. When an owner rejoins the overlay, its neighbors learn about it using the standard overlay mechanisms and then deliver their fragments to the final destination.

4.5 Fragment maintenance

Ideally, all N fragments of each object would be available in the network and stored on their respective fragment holders. However, there are various reasons why real Glacier installations may deviate from this ideal state: Nodes may miss fragment insertions due to short-term churn, key space ownership may change due to node joins and departures, and failures may cause some or all fragments stored on a particular node to be lost. To compensate for these effects, and to avoid a slow deterioration of redundancy, Glacier includes a *maintenance* mechanism.

Fragment maintenance relies on the fact that the placement function assigns fragments with similar keys to a similar set of nodes. If we assume for a moment that the nodeId distribution is perfectly uniform, each fragment holder has $N-1$ peers which are storing fragments of the exact same set of objects as itself. Then, the following simple protocol can be used:

1. The node compiles a list of all the keys (k, v) in its local fragment store, and sends this list to some of its peers.
2. Each peer checks this list against its own fragment store and replies with a list of manifests, one for each object missing from the list.
3. For each object, the node requests k fragments from its peers, validates each of the fragments against the manifest, and then computes the fragment that is to be stored locally.

With realistic nodeId distributions, the local portion of key space may not perfectly match that of the peer, so the node may have to divide up the list among multiple nodes. In very small networks, the placement function may even map more than one fragment to a single node, which must be accounted for during maintenance.

Glacier uses Bloom filters as a compact representation for the lists. To save space, these filters are parametrized such that they have a fairly high collision rate of about 25%, which means that about one out of four keys will not be detected as missing. However, the hash functions in the Bloom filter are changed after every maintenance cycle. Since maintenance is done periodically (typically once per hour), collisions cannot persist, and every fragment is eventually recovered.

4.6 Recovery

Glacier's maintenance process works whenever overlay communication is possible. Thus, the same mechanism covers normal maintenance and recovery after a large-scale failure. Compromised nodes either fail permanently, in which case other nodes take over their key segments, or they are eventually repaired and re-join the system with an empty fragment store. In both cases, the maintenance mechanism eventually restores full redundancy. Hence, there is no need for Glacier to explicitly detect that a correlated failure has occurred.

However, care must be taken to prevent congestive collapse during recovery. For this reason, Glacier limits the number of simultaneous fragment reconstructions to a fixed number R_{max}. Since the load spreads probabilistically over the entire network, the number of requests at any particular node is also on the order of R_{max}. Since Glacier relies on TCP for communication, this approach has the additional advantage of being self-clocking, i.e. the load is automatically reduced when the network is congested.

4.7 Garbage collection

When the lease associated with an object expires, Glacier is no longer responsible for maintaining its fragments

and may reclaim the corresponding storage. Since the lease is part of the authenticator, which accompanies every fragment, this process can be carried out independently by each storage node.

However, assuming closely synchronized clocks among the storage nodes would be unrealistic. Therefore, fragments are not deleted immediately; instead, they are kept for an additional *grace period* T_G, which is set to exceed the assumed maximal difference among the clocks. During this time, the fragments are still available for queries, but they are no longer advertised to other nodes during maintenance. Thus, nodes that have already deleted their fragments do not attempt to recover them.

Glacier has explicit protection against attacks on its time source, such as NTP. This feature is discussed in Section 6.

4.8 Configuration

Glacier's storage overhead is determined by the overhead for the erasure code, which is $\frac{N}{r}$, while the message overhead is determined by the number of fragments N that have to be maintained per object. Both depend on the guaranteed durability P_{min} and the maximal correlated failure fraction f_{max}, which are configurable.

Since suitable values for N and r have to be chosen *a priori*, i.e. before the failure has occurred, we do not know which of the nodes are going to be affected. Hence, all we can assume is that the unknown failure will affect any particular node with probability f_{max}. Note that this differs from the commonly assumed Byzantine failure model, where the attacker gets to choose the nodes that will fail. In our failure model, the attacker can only compromise nodes that share a common vulnerability, and these are distributed randomly in the identifier space because of the pseudo-random assignment of node identifiers.

Consider an object O whose N fragments are stored on N different nodes. The effect of the unknown correlated failure on O can be approximated by N Bernoulli trials; the object can be reconstructed if at least r trials have a positive outcome, i.e. with probability

$$D = P(s \geq r) = \sum_{k=r}^{N} \binom{N}{k} (1 - f_{max})^k \cdot {f_{max}}^{N-k}$$

The parameters N and r should be chosen such that P meets the desired level of durability. Figure 3 shows the lower bound on N and the storage overhead for different assumed values of f_{max} and for different choices of r. Table 1 shows a few example configurations.

While D represents the durability for an individual object, the user is probably more concerned about the durability of his entire collection of objects. If we assume that the number of storages nodes is large and that keys are assigned randomly (as is the case for content-hash keys), object failures are independent, and the probability that a collection of n objects survives the failure unscathed is $P_D(n) = D^n$. Figure 4 shows a graph of P_D for different values of D.

Failure f_{max}	Durability D	Code r	Fragments N	Storage S
0.30	0.9999	3	13	4.33
0.50	0.99999	4	29	7.25
0.60	0.999999	5	48	9.60
0.70	0.999999	5	68	13.60
0.85	0.999999	5	149	29.80
0.63	0.999999	1	30	30.00

Table 1. Example configurations for Glacier. For comparison, a configuration with simple replication (r=1) is included.

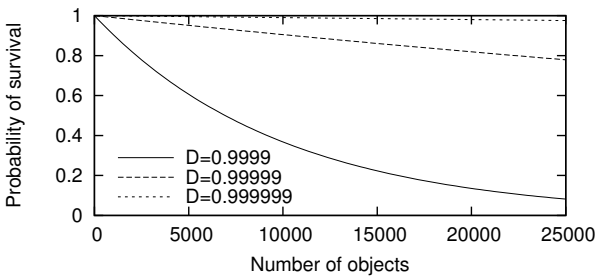


Figure 4. Probability of survival for collections of multiple objects.

If the value for f_{max} is accidentally chosen too low, Glacier still offers protection; the survival probability degrades gracefully as the magnitude of the actual failure increases. For example, if $f_{max} = 0.6$ and $P_{min} = 0.999999$ were chosen, P is still 0.9997 in a failure with $f = 0.7$, and 0.975 for $f = 0.8$. This is different in an introspective system, where an incorrect failure model can easily lead to a catastrophic data loss.

Another important parameter to consider is the lease time. If leases are short, then storage utilization is higher, since obsolete objects are removed more quickly; on the other hand, objects have to be refreshed more often. Clearly, the lease time must exceed both the maximal duration of a large-scale failure and the maximal absence of a user's node from the system. In practice, we recommend leases on the order of months. With shorter leases, users leaving for a long vacation might accidentally lose some of their data if they keep their machine offline during the entire time.

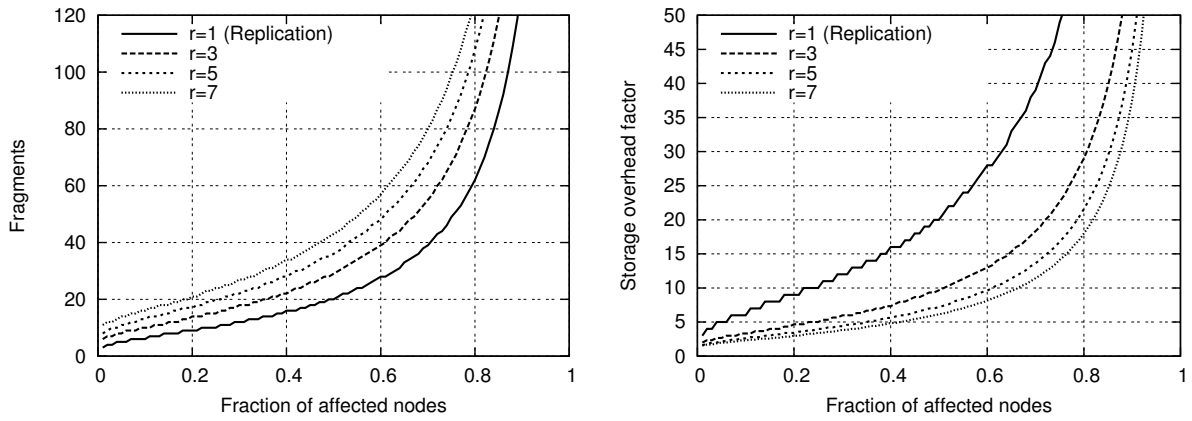


Figure 3. Number of fragments required for 99.9999% durability, and the resulting storage overhead.

5 Object aggregation

Glacier achieves data durability using massive redundancy. As a result, the number of internal objects Glacier must maintain is substantially larger than the number of application objects stored in Glacier. Each of these internal objects has a fixed cost; for example, each fragment is stored together with a manifest, and its key must be sent to other nodes during maintenance. To mitigate this cost, Glacier aggregates small application objects in order to amortize the cost of creating and maintaining fragments over a sufficient amount of application data.

In Glacier, each user is assumed to access the system through one node at a time. This node, which we call the user's *proxy*, holds the user's key material and is the only node in the system trusted by the user. All objects are inserted into Glacier from the object owner's proxy node. A user can use different proxy nodes at different times.

When a user inserts objects into Glacier, they are buffered at the user's proxy node. To ensure their visibility at other nodes, the objects are immediately inserted into Glacier's primary store, which is not aggregated. Once enough objects have been gathered or enough time has passed, the buffered objects are inserted as a single object into Glacier under an aggregate key. In the case of a proxy failure while an object is buffered, the next refresh operation will re-buffer the object for aggregation. Of course, buffered objects are vulnerable to large-scale correlated failures. If this is not acceptable, applications may invoke a `flush` method for important objects, which ensures that an aggregate with these objects is created and immediately stored in Glacier.

The proxy is also responsible for refreshing the owner's objects and for consolidating aggregates that contain too many expired objects. Performing aggregation and aggregate maintenance on a per-user basis avoids difficult problems due to the lack of trust among nodes. In return, Glacier foregoes the opportunity to bundle objects from different users in the same aggregate and to eliminate duplicate objects inserted by different users. In our experience, this is a small price to pay for the simplicity and robustness Glacier affords.

The proxy maintains a local *aggregate directory*, which maps application object keys to the key of the aggregate that contains the object. The directory is used when an object is refreshed and when an object needs to be recovered in response to an application request. After a failure of the proxy node, the directory needs to be regenerated from the aggregates. To do so, an owner's aggregates are linked in order of their insertion, forming a linked list, such that each aggregate contains the key of the previously inserted aggregate. The head of the list is stored in an application-specific object with a well-known key. To avoid a circularity, this object is *not* subject to aggregation in Glacier. The aggregate directory can be recovered trivially by traversing the list.

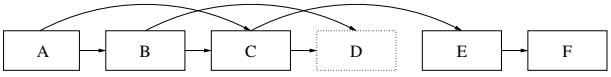


Figure 5. Reference graph. The object labeled 'D' has expired.

Aggregates are reclaimed in Glacier once all of the contained objects have expired. However, if aggregates expire in an order other than their insertion order, the aggregate list might become disconnected. To fix this problem, aggregates in the linked list may contain references to multiple other aggregates; thus, the aggregates actually form a directed acyclic graph (DAG, see Figure 5).

Glacier monitors the indegree of every aggregate in the DAG and tries to keep it above a fixed number d_{min}. If the indegree of an aggregate falls below this threshold, a pointer to it is added from the next aggregate to be inserted. This requires little extra overhead as long as insertions occur regularly; however, if a disconnection is imminent while no objects are inserted for an extended period of time, an empty aggregate may have to be created. This wastes a small amount of storage but, in our experience, occurs very rarely.

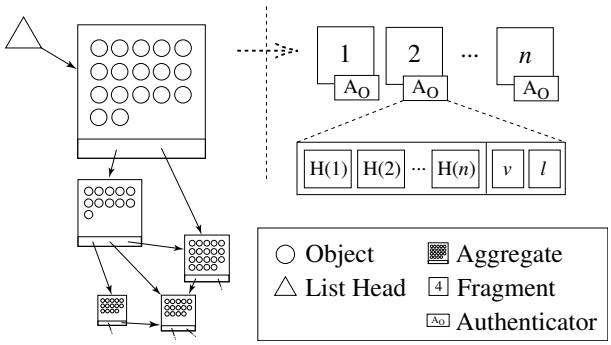


Figure 6. DAG of aggregates and the list head (left), and fragments of a single aggregate with the authenticator in detail (right).

An aggregate consists of tuples (o_i, k_i, v_i), where o_i is an object, k_i is the object's key, and v_i is the version number. Additionally, each aggregate contains one or more references to other aggregates. Note that the leases of the component objects are not stored; they are kept only in the aggregate directory, where they can be updated efficiently. The lease of the entire aggregate is the maximum of the component leases; for efficiency, Glacier tries to aggregate objects with similar leases.

5.1 Recovery

After a correlated failure, we must assume that all information that is not stored in Glacier is lost. In particular, this includes the contents of the primary store and, for most nodes, the aggregate directory.

The aggregate directory can be recovered by walking the DAG. First, the key of the most recently inserted aggregate is retrieved using a well-known key in Glacier. Then, the aggregates are retrieved in sequence and objects contained in each aggregate are added to the aggregate directory. The subleases of the component objects are set to the lease of the aggregate. Since aggregate leases are always higher than component leases, this is conservative. The primary store can either be repopulated lazily on demand by applications, or eagerly while walking the aggregate DAG.

Note that some of the references may be pointing to expired aggregates, so some of the queries issued to Glacier will fail. It is thus important to distinguish *actual* failures, in which at least $N-k+1$ fragment holders have been contacted but no fragments are found, from *potential* failures, in which some fragment holders are offline. In the latter case, recovery of the corresponding aggregate must be retried at a later time.

5.2 Consolidation

In order to maintain a low storage overhead, we use a mechanism similar to the segment cleaning technique in LFS [36]. Glacier periodically checks the aggregate directory for aggregates whose leases will expire soon, and decides whether to renew their leases. If the aggregate in question is small or contains many objects whose leases have already expired, the lease is not renewed. Instead, the non-expired objects are consolidated with new objects either from the local buffer or from other aggregates, and a new aggregate is created. The old aggregate is abandoned and its fragments are eventually garbage collected by the storing nodes.

Consolidation is particularly effective if object lifetimes are bimodal, i.e. if objects tend to be either short-lived or long-lived. By the time of the first consolidation cycle, the short-lived objects may have already expired, so the consolidated aggregate contains mostly long-lived objects. Such an aggregate then requires little maintenance, except for an occasional `refresh` operation.

6 Security

In this section, we discuss potential attacks against either the durability or the integrity of data stored in Glacier.

Attacks on integrity: Since Glacier does not have remote delete or update operations, a malicious attacker can only overwrite fragments that are stored on nodes under his control. However, each fragment holder stores a signed manifest, which includes an authenticator. Using this authenticator, fragment holders can validate any fragments they retrieve and replace them by other fragments if they do not pass the test. Assuming, as is customary, that SHA-1 is second pre-image resistant, generating a second fragment with the same hash value is computationally infeasible.

Attacks on durability: If an attacker can successfully destroy all replicas of an object in the primary store, as well as more than $n - r$ of its fragments, that object is lost. However, since there is no way to delete fragments remotely, the attacker can only accomplish this by either a targeted attack on the storage nodes, or indirectly

by interfering with Glacier's fragment repair or lease renewal. The former requires successful attacks on $n - r$ specific nodes within a short time frame, which is highly unlikely to succeed due to the pseudo-random selection of storage nodes. The latter cannot go unnoticed because Glacier relies on secure and authenticated overlay communication for fragment repair and lease renewal. This leaves plenty of time for corrective action by system administrators before too many fragments disappear due to uncorrelated failures, churn or lease expiration.

Attacks on the time source: Since the collection process is based on timestamps, an attacker might try to destroy an object by compromising a time source such as an NTP server and advancing the time beyond the object's expiration time. For this reason, storage nodes internally maintain all timestamps as relative values, translating them to absolute values only during shutdown and when communicating with another node.

Space-filling attacks: An attacker can try to consume all available storage by inserting a large number of objects into Glacier. While this does not affect existing data, no new data can be inserted because the nodes refuse to accept additional fragments. Without a remote deletion primitive, the storage can only be reclaimed gradually as the attacker's data expires. To prevent this problem, incentive mechanisms [32] can be added.

Attacks on Glacier: If a single implementation of Glacier is shared by all the nodes, Glacier itself must be considered as a potential source of failure correlation. However, data loss can result only due to a failure in one of the two mechanisms that actually delete fragments, handoff and expiration. Both are very simple (about 210 lines of code) and are thus unlikely to contain bugs.

Haystack-needle attacks: If an attacker can compromise his victim's personal node, he has, in the worst case, access to the cryptographic keys and can thus sign valid storage requests. Existing data cannot be deleted or overwritten; however, the attacker can try to make recovery infeasible by inserting decoy objects under existing keys, but with higher version numbers. The victim is thus forced to identify the correct objects among a gigantic number of decoys, which may be time-consuming or even infeasible.

However, notice that the attacker cannot compromise referential integrity. Hence, if the data structures are linked (as, for example, the aggregate log), the victim can recover them by guessing the correct key of a single object. One way to facilitate this is to periodically insert reference objects with well-known version numbers, such as the current time stamp. Thus, knowledge of the approximate time of the attack is sufficient to recover a consistent set of objects.

7 Experimental evaluation

To evaluate Glacier, we present the result of two sets of experiments. The first set is based on the use of Glacier as the storage layer for ePOST, a cooperative, serverless email system [28] that provides email service to a small group of users. ePOST has been in use for several months and it has used Glacier as its data store for the past 140 days. The second set is based on trace-driven simulations, which permit us to examine the system under a wider range of conditions, including a much larger workload corresponding to 147 users, up to $1,000$ nodes, a wider range of failure scenarios and different types of churn.

The Glacier prototype is built on top of the FreePastry [21] implementation of the Pastry [37] structured overlay and makes use of the PAST [20] distributed hash table service as its primary store. Since the ePOST system relies on PAST for storage, Glacier now provides durable storage for ePOST.

7.1 ePOST experiments

Over time, our experimental ePOST overlay grew from 20 to currently 35 nodes. The majority of these nodes are desktop PCs running Linux; however, there are also machines running OS X and Windows. Our user base consists of 8 *passive* users, which are still primarily using server-based email but are forwarding their incoming mail to the ePOST overlay, and 9 *active* users, which rely on ePOST as their main email system.

ePOST uses Glacier with aggregation to store email and the corresponding metadata. For each object, Glacier maintains $N = 48$ fragments using an erasure code with $r = 5$, i.e. any five fragments are sufficient to restore the object. In this configuration, each object can survive a correlated failure of $f_{max} = 60\%$ of all nodes with probability $P_{min} = 0.999999$. We are aware of the fact that with only 35 nodes, our experimental deployment is too small to ensure that fragment losses are uncorrelated. Nevertheless, we chose this configuration to get realistic numbers for the per-node overhead.

Each of the nodes in the overlay periodically writes statistics to its log file, including the number of objects and aggregates it maintains, the amount of storage consumed locally, and the number and type of the messages sent. We combined these statistics to obtain a view of the entire system.

While working with Glacier and ePOST, we were able to collect much practical experience with the system. We had to handle several node failures, including kernel panics, JVM crashes and a variety of software problems and configuration errors. Also, there were some large-scale correlated failures; for instance, a configuration er-

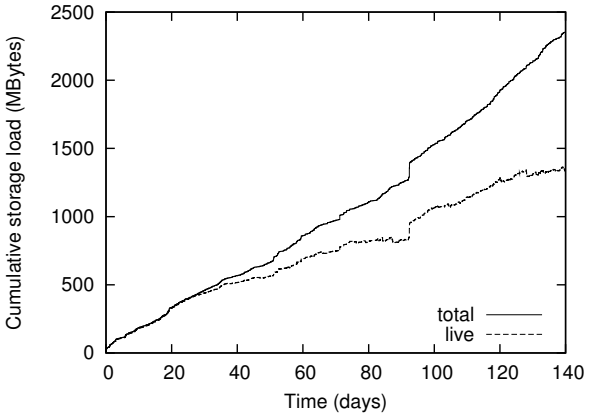


Figure 7. Storage load in ePOST.

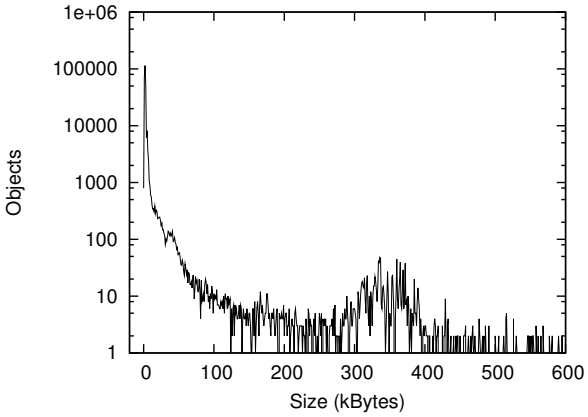


Figure 8. Object sizes in ePOST.

ror once caused an entire storage cluster of 16 nodes to become disconnected. Glacier was able to handle all of these failures. Also, note that Glacier was still under active development when it was deployed. During our experiments, we actually found two bugs, which we were able to fix simply by restarting the nodes with the updated software.

We initially configured Glacier so that it would consider nodes to have failed if they did not join the overlay for more than 5 days. However, it turned out that some of the early ePOST adopters started their nodes only occasionally, so their regions of key space were repeatedly taken over by their peers and their fragments reconstructed. Nevertheless, we decided to include these results as well because they show how Glacier responds to an environment that is heavily dynamic.

7.2 Workload

We first examined the workload generated by ePOST in our experimental overlay. Figure 7 shows the cumulative size of all objects inserted over time, as well as the size of the objects that have not yet expired. Objects are inserted with an initial lease of one month and are refreshed every day until they are no longer referenced.

Figure 8 shows a histogram of the object sizes. The histogram is bimodal, with a high number of small objects ranging between $1 - 10$kB, and a lower number of large objects. Out of the $274,857$ objects, less than 1% were larger than 600kB (the maximum was 9.1MB); these are not shown for readability. The small objects typically contain emails and their headers, which are stored separately by ePOST, while the large objects contain attachments. Since most objects are small compared to the fixed-size manifests used by Glacier (about 1kB), this indicates that aggregation can considerably increase storage efficiency.

7.3 ePOST storage

Next, we looked at the amount of storage required by Glacier to store the above workload. Figure 9 shows the combined size of all fragments in the system. The storage grows slowly, as new email is entering the system; at the same time, old email and junk mail is deleted by the users and eventually removed by the garbage collector.

In this deployment, garbage is not physically deleted but rather moved to a special trash storage, whose size is also shown. We used a lease time of 30 days for all objects. For compatibility reasons, ePOST maintains its on-disk data structures as gzipped XML. On average, this creates an additional overhead of 32%, which is included in the figures shown.

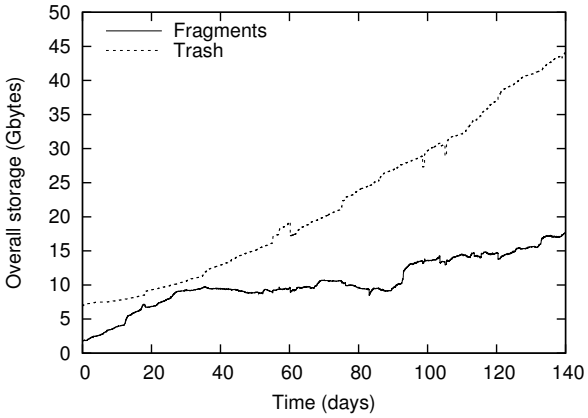


Figure 9. Storage consumed by Glacier fragments and trash.

Figure 10 compares the size of the on-disk data structures to the actual email payload. It shows the average number of bytes Glacier stored for each byte of payload, excluding trash, but including the 32% overhead from

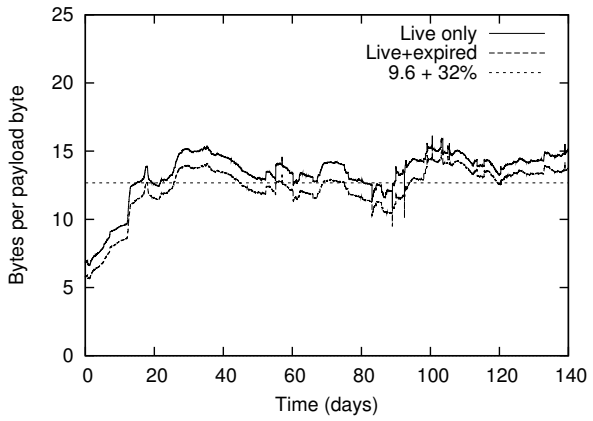


Figure 10. Storage factor, including serialization overhead.

XML serialization, for live data and for all data stored in Glacier. The average storage overhead over time is very close to the expected factor of 9.6 plus the 32% due to XML serialization.

7.4 ePOST traffic

Figure 11 shows the average traffic generated by an ePOST node in bytes and in Pastry-level messages sent per minute (the messages are sent over TCP, so small messages may share a single packet, and large messages may require multiple packets). For comparison, we also report traffic statistics for the other subsystems involved in ePOST, such as PAST and Scribe [11].

The traffic pattern is heavily bimodal. During quiet periods (e.g. days 30-50), Glacier generally sends fewer messages than PAST because it can mask short-term churn, but since the messages are larger because of the difference in storage factors (9.6 versus 3), the overall traffic is about the same. In periods with a lot of node failures (e.g. days 80-120), Glacier must recover the lost fragments by reconstructing them from other fragments, which creates additional load for a short time. The increase in Pastry traffic on day 104 was caused by an unrelated change in Pastry's leaf set stabilization protocol.

The traffic generated by Glacier can be divided into five categories:

- **Insertion:** When new objects are inserted, Glacier identifies the fragment holders and transfers the fragment payload to them.
- **Refresh:** When the leases for a set of objects are extended, Glacier sends the updated part of the storage manifest to the current fragment holders.
- **Maintenance:** Peer nodes compare their key lists, and lost fragments are regenerated from other fragments.
- **Handoff:** Nodes hand off some of their fragments to a new node who has taken over part of their key space.
- **Lookup:** Aggregates are retrieved when an object is lost from the object cache, or when small aggregates are consolidated into larger ones.

In Figure 12, the Glacier traffic is broken down by category. In times with a low number of failures, the traffic is dominated by insertions and refreshes. When the network is unstable, the fraction of handoff and maintenance traffic increases. In all cases, the maintenance traffic remains below 15 packets per host and minute, which is very low.

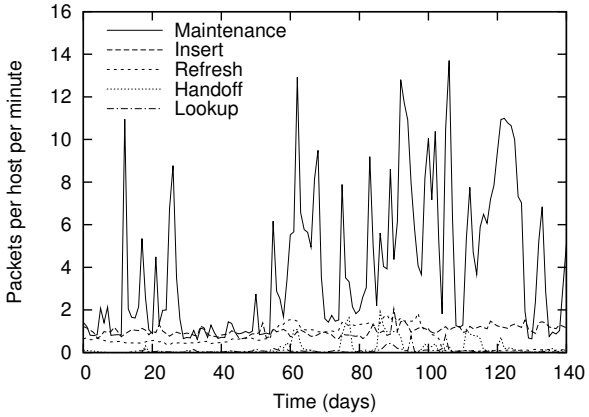


Figure 12. Messages sent by Glacier, by activity.

7.5 ePOST aggregation

To determine the effectiveness of aggregation, we also collected statistics on the number of objects and aggregates in the system. We distinguished between *live* objects, whose lease is still valid, and *expired* objects, which are still stored as part of an aggregate but are eligible for garbage collection.

Figure 13 shows the average number of objects in each aggregate. In our system, aggregation reduced the number of keys by more than an order of magnitude. Moreover, our results show that the number of expired objects remains low, which indicates that aggregate consolidation is effective.

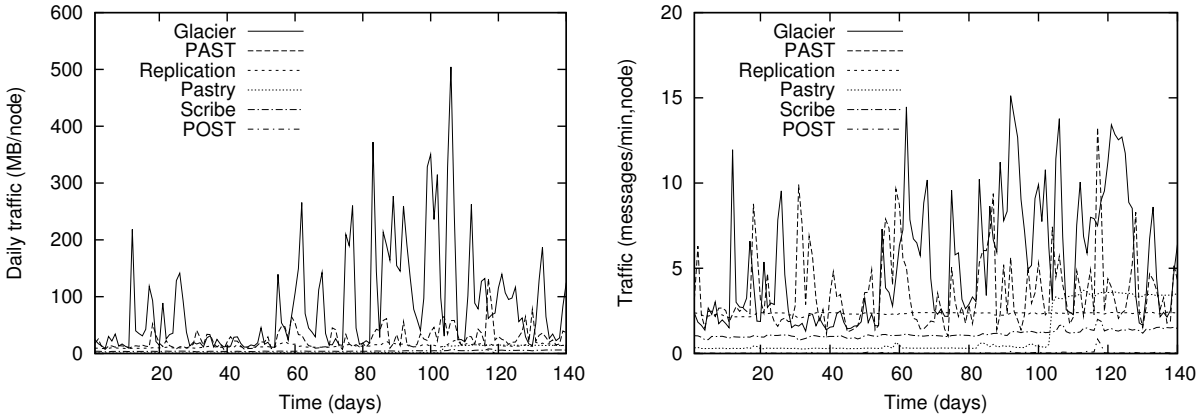


Figure 11. Average traffic per node and day (left) and average number of messages per node and minute (right).

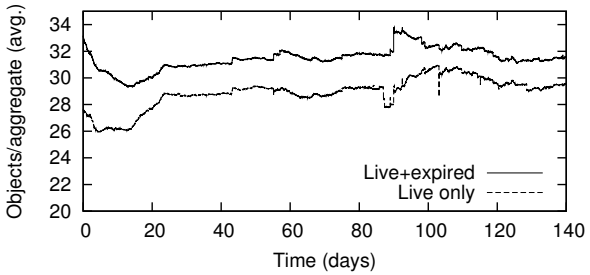


Figure 13. Aggregation factor.

7.6 ePOST recovery

To study Glacier's behavior in the event of a large-scale correlated failure, we randomly selected 13 of the 31 nodes in our experimental ePOST overlay and copied their local fragment store to 13 fresh nodes (note that, since our overlay has fewer than $N = 48$ nodes, some nodes store more than one fragment of the same object). The primary PAST store and the metadata were not copied. We then started a new Pastry overlay with only these 13 nodes. The resulting situation corresponds to a 58% failure in the main overlay, which is close to our assumed $f_{max} = 60\%$.

We then completely re-installed ePOST on a fourteenth node and let it join the ring. One of the authors entered his email address and an approximate date when he had last used ePOST. From this information, ePOST first determined the key of its metadata backup in Glacier by hashing the email address; then it retrieved the backup and extracted from it the root key of the aggregate DAG. The aggregation layer then reconstructed the DAG and restored the objects in it to the primary store. This process took approximately one hour to complete but could be sped up significantly by adding some simple optimizations. Afterwards, ePOST was again ready for use; all data that had been stored using Glacier was fully recovered.

7.7 Simulations: Diurnal behavior

For this and the following experiments, we used a trace-driven simulator that implements Glacier and the aggregation layer. Since we wanted to model a system similar to ePOST, we used a trace from our department's email server, which contains $395,350$ delivery records over a period of one week (09/15-09/21). Some email is carbon-copied to multiple recipients; we delivered each copy to a separate node, for a total of $1,107,504$ copies or approximately 8 GBytes. In the simulation, Glacier aggregates of up to 100 objects using a simple, greedy first-fit policy.

In our first simulation, we explore the impact of diurnal short-term churn. In their study of a large deployment of desktop machines, Bolosky et al. [8] report that the number of available machines, which was generally around approximately $45,000$, dropped by about $2,500$ (5.5%) at night time and by about $5,000$ (11.1%) during weekends. In our simulations, we modeled a ring of 250 nodes with the behavior from the study, where $M\%$ of the nodes are unavailable between 5pm and 7am on weekdays and $2M\%$ on weekends. The experiment was run for one week of simulation time, starting from Wednesday, 09/15, and the entire trace was used. Glacier was configured with the maximum offline time T_{max} set to one week.

Figure 14 shows how this behavior affects the total message overhead, which includes all messages sent over the entire week, for different values of M. As churn increases, fewer fragments can be delivered directly to their respective fragment holders, so insertion traffic de-

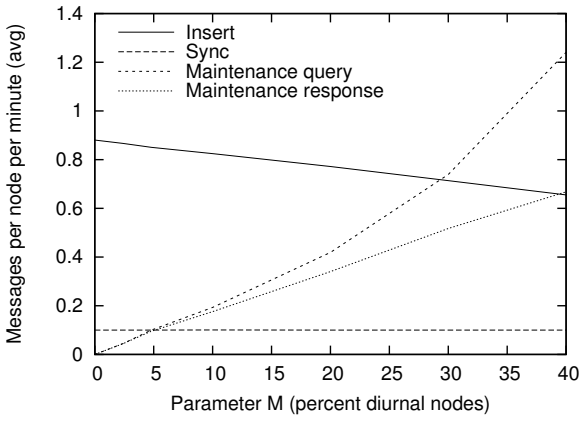


Figure 14. Impact of diurnal short-term churn on message overhead.

creases. However, the lost fragments must be recovered when the fragment holders come back online, so the maintenance overhead increases. As an additional complication, the probability that fragments are available at the designated fragment holder decreases, so maintenance requires more attempts to successfully fetch a fragment. This causes the disparity between maintenance requests and replies, which are shown separately in the figure.

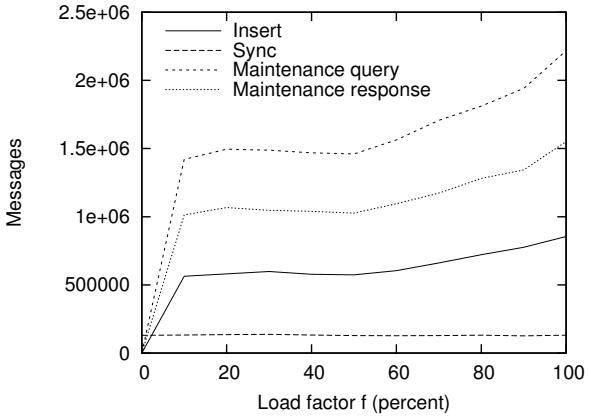


Figure 15. Impact of increasing load on message overhead.

7.8 Simulation: Load

In our second simulation, we study how the load influences the message overhead. We again used a overlay of 250 nodes and the trace from our mail server, but this time, we used only a fraction f of the messages. Instead of diurnal churn, we simulated uncorrelated short-term churn with a mean session time of 3 days and a mean pause time of 16 hours, as well as long-term churn with a mean node lifetime of 8 days. We varied the parameter f between 0 and 1.

Figure 15 shows how the load influences the cumulative message overhead over the entire week. Under light load, the message overhead remains approximately constant. This is because aggregates are formed periodically by every node, even if less than 100 objects are available in the local buffer. As the load increases further, the increase in overhead is approximately linear, as expected.

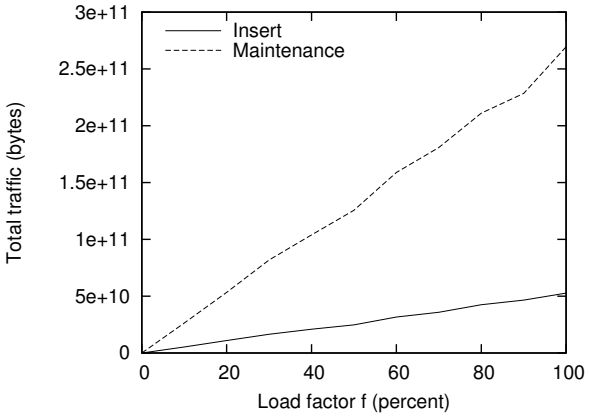


Figure 16. Impact of increasing load on total traffic.

Figure 16 shows the same overhead in bytes. Here, the threshold effect does not appear. Also, note the high maintenance overhead, as expected. This is due to the aggressive parameters we used for churn; at a node lifetime of eight days, almost all the nodes are replaced at least once during the simulation period, their local fragment store being fully regenerated every time. For their desktop environment, Bolosky et al. [8] report an expected machine lifetime of 290 days and low short-term churn, which would reduce the maintenance overhead considerably.

7.9 Simulation: Scalability

In our third simulation, we examine Glacier's scalability in terms of the number of participating nodes. We used the same trace as before, but scaled it such that the storage load per node remained constant; the full trace was used for our maximum setting of $N = 1000$ nodes. The churn parameters are the same as before.

Figure 17 shows the message overhead per node for different overlay sizes. As expected, the net overhead remains approximately constant; however, since query messages are sent using the Pastry overlay, the total number of messages grows slowly with $N \; log \; N$.

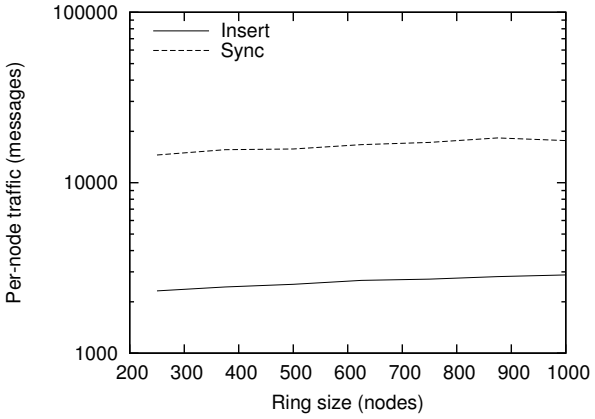


Figure 17. Message overhead for different overlay sizes and a constant per-node storage load.

7.10 Discussion

The storage overhead required to sustain large-scale correlated failures is substantial. In our experiments, we used fairly aggressive parameters ($f_{max} = 60\%$, $P_{min} = 0.999999$), which resulted in an 11-fold storage overhead. However, this cost is mitigated by the fact that Glacier can harness vast amounts of underutilized storage that is unreliable in its raw form. Moreover, only truly important and otherwise unrecoverable data must be stored in a high-durability Glacier store and is thus subject to large storage overhead. Data of lesser importance and data that can be regenerated after a catastrophic failure can be stored with far less overhead in a separate instance of Glacier that is configured with a less stringent durability requirement.

On the other hand, our experiments show that Glacier is able to manage this large amount of data with a surprisingly low maintenance overhead and that it is scalable both with respect to load and system size. Thus, it fulfills all the requirements for a cooperative storage system that can leverage unused disk space and provide hard, analytical durability guarantees, even under pessimistic failure assumptions. Moreover, our experience with the ePOST deployment shows that the system is practical, and that it can effectively protect user data from large-scale correlated failures. The ever-increasing number of virus and worm attacks strongly suggests that this property is crucial for cooperative storage system.

8 Conclusions

We have presented the design and evaluation of Glacier, a system that ensures durability of unrecoverable data in a cooperative, decentralized storage system, despite large-scale, correlated, Byzantine failures of storage nodes. Glacier's approach is 'extreme' in the sense that it does not rely on introspection, which has inherent limitations in its ability to capture all sources of correlated failures; instead, it uses massive redundancy to mask the effects of large-scale correlated failures such as worm attacks. The system uses erasure codes and garbage collection to mitigate the storage cost of redundancy and relies on aggregation and a loosely coupled fragment maintenance protocol to reduce the message costs. Our experience with a real-world deployment shows that the message overhead for maintaining the erasure coded fragments is low. The storage overheads can be substantial when the availability requirements are high and a large fraction of nodes is assumed to suffer correlated failures. However, cooperative storage systems harness a potentially huge amount of storage. Glacier uses this raw, unreliable storage to provide hard durability guarantees, which is required for important and otherwise unrecoverable data.

9 Acknowledgements

This paper has benefited considerably from the comments of the anonymous NSDI reviewers and our shepherd, Mike Dahlin. This research was supported in part by Texas ATP (003604-0079-2001), by NSF (ANI-0225660, http://project-iris.net) and by gifts from Microsoft Research and from Intel Research.

References

[1] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, John Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. OSDI*, Boston, MA, Dec 2002.

[2] Noga Alon, Haim Kaplan, Michael Krivelevich, Dahlia Malkhi, and Julien Stern. Scalable secure storage when half the system is faulty. In *Proc. ICALP*, 2000.

[3] Yair Amir and Avishai Wool. Evaluating quorum systems over the internet. In *Proc. 26th Annual Intl. Symposium on Fault-Tolerant Computing (FTCS '96)*, page 26. IEEE Computer Society, 1996.

[4] Mehmet Bakkaloglu, Jay J. Wylie, Chenxi Wang, and Gregory R. Ganger. On correlated failures in survivable storage systems. Technical Report Carnegie Mellon CMU-CS-02-129, May 2002.

[5] Ranjita Bhagwan, Kiran Tati, Yuchung Cheng, Stefan Savage, and Geoffrey M. Voelker. TotalRecall: System support for automated availability management. In *Proc. NSDI*, San Francisco, CA, Mar 2004.

[6] Charles Blake and Rodrigo Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Proc. HotOS*, pages 1–6, Lihue, HI, May 2003.

[7] Johannes Bloemer, Malik Kalfane, Marek Karpinski, Richard Karp, Michael Luby, and David Zuckerman. An XOR-based erasure-resilient coding scheme. Technical Report ICSI TR-95-048, Aug 1995.

[8] William J. Bolosky, John R. Douceur, David Ely, and Marvin

Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proc. Intl. Conference on Measurement and Modeling of Computer Systems*, Santa Clara, CA, June 2000.

[9] Ali Raza Butt, Troy A. Johnson, Yili Zheng, and Y. Charlie Hu. Hoard: A peer-to-peer enhancement for the network file system. Technical Report Purdue University ECE-03-08, 2003.

[10] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proc. OSDI*, Boston, MA, Dec 2002.

[11] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE JSAC*, 20(8), October 2002.

[12] Josh Cates. Robust and efficient data management for a distributed hash table. Master's thesis, Massachusetts Institute of Technology, May 2003.

[13] Fay W. Chang, Minwen Ji, Shun-Tak A. Leung, John MacCormick, Sharon E. Perl, and Li Zhang. Myriad: Cost-effective disaster tolerance. In *Proc. FAST*, pages 103–116. USENIX Association, 2002.

[14] Yuan Chen, Jan Edler, Andrew Goldberg, Allan Gottlieb, Sumeet Sobti, and Peter Yianilos. A prototype implementation of archival intermemory. In *Proc. 4th ACM Conference on Digital Libraries (DL'99)*, 1999.

[15] Landon P. Cox, Christopher D. Murray, and Brian D. Noble. Pastiche: Making backup cheap and easy. In *Proc. OSDI*, Boston, MA, Dec 2002.

[16] Frank Dabek, Emma Brunskill, M. Frans Kaashoek, David Karger, Robert Morris, Ion Stoica, and Hari Balakrishnan. Building peer-to-peer systems with chord, a distributed lookup service. In *Proc. HotOS*, Schloss Elmau, Germany, May 2001.

[17] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proc. SOSP*, Banff, Canada, Oct 2001.

[18] Roger Dingledine, Michael J. Freedman, and David Molnar. The Free Haven project: Distributed anonymous storage service. In *Proc. Workshop on Design Issues in Anonymity and Unobservability (LNCS 2009)*, Jul 2000.

[19] John R. Douceur. The Sybil attack. In *Proc. of IPTPS*, Mar 2002.

[20] Peter Druschel and Anthony Rowstron. PAST: a large-scale, persistent peer-to-peer storage utility. In *Proc. HotOS*, Schloss Elmau, Germany, May 2001.

[21] The FreePastry web site. http://freepastry.rice.edu/.

[22] Edward Grochowski. Emerging trends in data storage on magnetic hard disk drives. *Datatech*, pages 11–16, Sep 1998.

[23] Flavio Junqueira, Ranjita Bhagwan, Alejandro Hevia, Keith Marzullo, and Geoffrey M. Voelker. Surviving internet catastrophes. In *Proc. 2005 Usenix Annual Technical Conference*, Apr 2005.

[24] Flavio Junqueira, Ranjita Bhagwan, Keith Marzullo, Stefan Savage, and Geoffrey M. Voelker. The Phoenix recovery system: Rebuilding from the ashes of an internet catastrophe. In *Proc. HotOS*, Lihue, HI, May 2003.

[25] Flavio Junqueira and Keith Marzullo. Designing algorithms for dependent process failures. In *Proc. Intl. Workshop on Future Directions in Distributed Computing (FuDiCo)*, 2002.

[26] Kimberly Keeton, Cipriano Santos, Dirk Beyer, Jeff Chase, and John Wilkes. Designing for disasters. In *Proc. FAST*, Mar 2004.

[27] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proc. ASPLOS*, Nov 2000.

[28] Alan Mislove, Ansley Post, Charles Reis, Paul Willmann, Peter Druschel, Dan S. Wallach, Xavier Bonnaire, Pierre Sens, Jean-Michel Busca, and Luciana Arantes-Bezerra. POST: A secure, resilient, cooperative messaging system. In *Proc. HotOS*, Lihue, HI, May 2003.

[29] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the Slammer worm. *IEEE Security and Privacy*, 1(4):33–39, Jul 2003.

[30] David Moore, Colleen Shannon, and Jeffery Brown. Code-Red: A case study on the spread and victims of an internet worm. In *Proc. 2nd ACM Internet Measurement Workshop*, 2002.

[31] Athicha Muthitacharoen, Robert Morris, Thomer Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proc. OSDI*, Boston, MA, Dec 2002.

[32] Tsuen-Wan Ngan, Dan S. Wallach, and Peter Druschel. Enforcing fair sharing of peer-to-peer resources. In *Proc. IPTPS*, Berkeley, CA, Feb 2003.

[33] Network time protocol (version 3) specification, implementation and analysis. RFC 1305.

[34] Onestat.com. http://www.onestat.com/.

[35] Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[36] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proc. SOSP*, pages 1–15. ACM Press, 1991.

[37] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. Middleware*, pages 329–350, Heidelberg, Germany, Nov 2001.

[38] Eugene H. Spafford. The internet worm incident. Technical Report Purdue CSD-TR-933, West Lafayette, IN 47907-2004, 1991.

[39] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to 0wn the internet in your spare time. In *Proc. USENIX Security*, Aug 2002.

[40] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. ACM SIGCOMM*, pages 149–160, San Diego, CA, Aug 2001.

[41] Dong Tang and Ravishankar K. Iyer. Analysis and modeling of correlated failures in multicomputer systems. *IEEE Trans. Comput.*, 41(5):567–577, 1992.

[42] Hakim Weatherspoon and John Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proc. IPTPS*, MIT Faculty Club, Cambridge, MA, Mar 2002.

[43] Hakim Weatherspoon, Tal Moscovitz, and John Kubiatowicz. Introspective failure analysis: Avoiding correlated failures in peer-to-peer systems. In *Proc. International Workshop on Reliable Peer-to-Peer Distributed Systems*, Oct 2002.

[44] Chris Wells. The OceanStore archive: Goals, structures, and self-repair. U.C. Berkeley Masters Report, May 2002.

[45] Jay J. Wylie, Michael W. Bigrigg, John D. Strunk, Gregory R. Ganger, Han Kiliccote, and Pradeep K. Khosla. Survivable information storage systems. *IEEE Computer*, 33(8):61–68, Aug 2000.

[46] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report U.C. Berkeley UCB//CSD-01-1141, Apr 2001.

[47] Cliff Changchun Zou, Weibo Gong, and Don Towsley. Code red worm propagation modeling and analysis. In *Proc. 9th ACM Conference on Computer and Communication Security (CCS'02)*, Washington, DC, Nov 2002.

Quorum: Flexible Quality of Service for Internet Services

Josep M. Blanquer, Antoni Batchelli, Klaus Schauser and Rich Wolski
Department of Computer Science, University of California Santa Barbara
{blanquer,tbatchel,schauser,rich}@cs.ucsb.edu

Abstract

In this paper we describe Quorum, a non-invasive approach to scalable quality-of-service provisioning that uses traffic shaping, admission control, and response monitoring at the border of an Internet site to ensure throughput and response time guarantees.

We experimentally compare an implementation of Quorum both to hardware over-provisioning and to leading software approaches using real world workloads. Our results show that Quorum can enforce the same QoS guarantees as either of the compared approaches, while achieving better resource utilization than over-provisioning and without the application rewriting overhead required by intrusive software approaches. We also demonstrate that our implementation can successfully handle extreme situations such as sudden traffic surges, application misbehavior and node failures. Furthermore, we demonstrate the flexibility of Quorum by providing QoS guarantees for a complex and heterogeneous Internet service that cannot be implemented by other current software approaches.

1 Introduction

The current commercial importance of Internet services makes it imperative for companies relying on web-based technologies to offer and guarantee predictable, consistent, and differentiated quality of service (QoS) to their consumers. For example, e-commerce companies often want to provide faster response times for purchasing than for catalog browsing to ensure that no sale is lost due to the perception of an unresponsive transaction. Differentiated QoS also enables more general and flexible application hosting environments. For example, a service provider that hosts a personalized webmail portal for several companies wants to guarantee different levels of service to its customers and to ensure that these service guarantees are provided to each customer independently, regardless of overload or misbehavior of the others.

To meet large demand, scalable Internet services are commonly hosted using clustered architectures where a number of machines, rather than a single server, work together in a distributed and parallel manner to serve requests. Delivering reliable service quality guarantees in this distributed setting is the difficult challenge that our work addresses.

Both research and commercial Internet service communities have explored hardware-based and software-based approaches to QoS provisioning. The state-of-the-practice in current commercial settings is to deploy independent clusters for each service (hardware partitioning), each of which comprises enough capacity to service worst-case load conditions (over-provisioning). Unfortunately, because load fluctuations can be substantial, hardware partitioning and over-provisioning incurs a potentially high cost (sufficient resources must be available in each partition to handle load spikes) and low resource utilization (the extra resources are idle between spikes), making this approach inefficient.

As a result, software-based approaches have been proposed and developed to make better use of the resources employed to host Internet services. These approaches focus on embedding QoS logic at different levels of the site's internal software, including operating system [4, 6, 10, 37], middleware [33, 34, 41], and application code [2, 8, 40]. It is the function of this logic to distribute, effectively, the workload among the cluster resources as a way of improving both resource utilization and client experience. Low-level techniques have been shown to provide a tight control on the utilization of resources (e.g., disk bandwidth or processor usage) while techniques that are closer to the application layer are able to satisfy QoS requirements that are more directly experienced by clients. However, these software solutions require the hosted application services and/or the hosting operating system to be customized for QoS provisioning, thereby limiting flexibility and extensibility. Furthermore, most current Internet sites include a myriad of different hardware and software platforms which are constantly evolving and changing. An invasive QoS solution that requires the reprogramming of hosted service code carries with it high development and testing costs when new services are introduced, or the existing

site components (hardware and software) are reconfigured, upgraded, extended, etc. More problematically, the source code for many service components hosted at a site may not be available for proprietary reasons. This lack of source code makes the necessary software reprogramming remarkably difficult. Thus the growing complexity associated with Internet service hosting in commercial settings makes intrusive software QoS strategies less attractive as the need for extensibility and flexibility increases.

To address these needs, we propose a new approach to QoS provisioning for Internet services. Our approach offers reliable QoS guarantees at a lower cost than state-of-the-practice techniques, while giving the service providers the much needed flexibility that they require to rapidly reconfigure, upgrade and extend their complex set of services. In this paper we present *Quorum*, a non-invasive software approach that treats the cluster and the services it is hosting as a "black-box" system and uses only feedback-driven techniques to control dynamically which and when each of the requests from the clients is forwarded into the cluster. Because traffic shaping and admission control is done at the entrance of the site, and the system uses only the observed request and response streams for its control algorithms, new services can be added, old ones upgraded, and resources reconfigured without re-engineering the necessary QoS mechanisms into the services themselves or the system software that supports them.

We report on an implementation of Quorum and its experimental comparison with the state-of-the-practice (i.e., over-provisioning) and state-of-the-art (e.g., Neptune [33, 34]) software solutions using realistic services, client request traces and clustered machines. Neptune is a research and now commercially successful middleware system that implements QoS for Internet services, but which requires the services themselves to be re-written to use Neptune primitives. Using the *Teoma* [35] search engine, which is explicitly programmed so it can use Neptune, we show that Quorum can enforce the same QoS guarantees as Neptune for Neptune-enabled services, but without the additional engineering overhead associated with modifying the services that it supports. Furthermore, we illustrate Quorum's ability to handle extreme situations such as sudden traffic surges, or internal application misbehavior – capabilities that are necessary for a successful deployment in large-scale, realistic settings. We also demonstrate the flexibility of Quorum by showing how it can provide QoS guarantees for complex heterogeneous Internet services which cannot be modified – a capability that none of the published, pre-existing software approaches is capable of achieving at present.

1.1 Contributions

This paper makes five main contributions:

- We present Quorum as a novel approach to QoS provisioning for large-scale Internet services that uses only observed input request and output response streams to control the load within the site so that quality guarantees are met.
- We describe a working implementation and demonstrate its viability using a large cluster system hosting commercial and community benchmark Internet services.
- We compare Quorum with the state-of-the-practice and state-of-the-art approaches in terms of efficiency and the degree to which they maintain QoS guarantees for both throughput and response times.
- We show the robustness of Quorum in successfully overcoming extreme situations (i.e., sudden traffic surges, application misbehavior and node failures) which arise in current commercial settings.
- We demonstrate that the flexibility provided by Quorum enables more efficient deployments of complex, heterogeneous Internet services than can currently be supported by existing approaches.

The remainder of this paper is organized as follows. Section 2 defines the models and assumptions employed by our approach and formally states the terms in which the QoS challenge is defined. Section 3 introduces Quorum's approach and further describes its architecture. Section 4 experimentally compares Quorum to the best of the known approaches. In Section 5 we demonstrate the robustness of Quorum under extreme situations and also show its flexibility in providing reliable QoS guarantees in complex heterogeneous services. In Section 6 we discuss related work, and we conclude in Section 7.

2 Defining the QoS Challenge

Before describing the architecture of Quorum we define the models and assumptions employed by our approach, as well as detail the terms by which the QoS challenge is defined. We begin by outlining the model of Internet service transactions we use. We treat Internet services (see Figure 1) as a stream of requests coming from clients that are received at the entrance of the site, computed by the internal resources, and returned back to the clients upon completion. In the case of system overload or internal error condition, requests can be dropped before completion and thus may not be returned to the client. Each of the

requests that are received from the clients can be classified or grouped into different *service classes* according to a combination of service type and client identity.

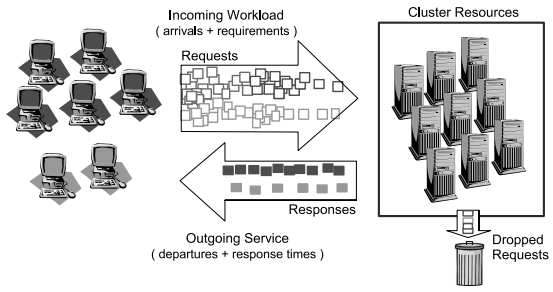


Figure 1: System model for Internet services.

The computation of requests is modeled by treating the cluster as a parallel and multi-level resource system that processes requests in a time-shared manner. More specifically we model the cluster site as a black-box system that has the following two properties: 1) unbiased treatment: any request entering the cluster will be computed with the same priority, and 2) time-multiplexed: the internal computation is done in a multiplexed way where requests interleave the usage of resources in time intervals that are short, relative to the response time guarantees. The simplicity of our model allows cluster systems to be treated analytically, while remaining powerful enough to capture the behavior of most time-shared systems and thus the majority of existing sites. Note that these properties are defined in terms of the overall cluster behavior and may not necessarily hold true for each of the internal resources individually.

We define the QoS challenge as the ability to guarantee, at all times, a predefined quantitative characterization of the traffic in each service class as measured at the *output* of the cluster. Such traffic characterization is expressed through a QoS policy, which contains the desired QoS guarantees for each of the participating service classes. Such quality guarantees are defined at the boundary of the cluster and do not extend to requests traversing the internet back to the end users. Our goal is not to provide end-to-end guarantees as we see network QoS as a complementary function. We consider the quality characteristics defined in the QoS policy to be specified in terms of statistical (or soft) guarantees.

Finally, our system requires that the specified QoS policy that must be enforced is, in fact, *feasible* for the expected workloads and cluster capacity. A QoS policy is feasible if the existing cluster can meet the QoS guarantees without requiring any QoS mechanisms (i.e., a simple load-balancer) subject to the following two conditions: 1) incoming rates for each class are always kept below their guaranteed throughput and 2) resource demands of incoming requests do not surpass their expected computation requirements. The expected computation requirements for a request stream are dependent on the type of service offered and must be agreed upon, *a priori*, between the provider and the consumer. In fact, feasibility is an implicit test or calculation that any service provider must perform when dimensioning their cluster for a given expected demand. Note feasibility has already taken into account the software and hardware configuration of the cluster, as well as any possible internal bottlenecks that may occur for the expected workloads. Since Quorum depends only on the feasibility condition, it can continue to operate correctly regardless of the presence of internal cluster bottlenecks.

3 The Quorum Architecture

In this section we describe the architecture of Quorum that follows our previously stated model and assumptions. In Quorum, the QoS policy is specified as a list of *QoS classes* describing the quality that must be ensured for each class of service. We define a QoS class as a tuple that describes: 1) how to identify requests of this class (*classification rules*) and, 2) what type of QoS must be enforced (*output guarantees*). In the same way as level-7 load-balancers [19, 20, 29], Quorum classifies requests based on a combination of parameters such as IP address, port, URL and path. Output guarantees are specified in terms of guaranteed minimum throughput and maximum response time. For example, Table 1 describes a QoS policy containing two QoS classes for a service provider hosting webmail portals for two different companies. In the example, *BigCorp* has a much higher guaranteed throughput due to an expected higher traffic volume and *SmallCorp* requires much tighter response time guarantees for its users. Notice that the definition of output guarantees includes both throughput and response time requirements. While it is often possible to meet one type of guarantee at the expense of the other, our solution accommodates both. Additionally, Quorum allows throughput and response time guarantees to be expressed using either percentiles or averages since the way in which each customer wishes to view a guarantee varies. In both cases, however, the time frame over which the average or percentile is computed is substantially longer than the time required to service an individual request.

Quorum uses a single-policy enforcement engine to intercept and control in-bound traffic at the entrance of the site hosting the services. By tracking the responses to requests that are served within the site, our system automatically determines when new requests can be allowed entry such that a specified set of QoS guarantees will be enforced. No knowledge of the internals of the

QoS Class	Classification	Output Guarantees	
	Layer 7 pattern	Throughput (Avg. req/s)	Response Time (95th% ms)
BigCorp	`http://bigcorp.com/mail/*`	800	1000
SmallCorp	`http://smallcorp.com/mail/*`	50	200

Table 1: Example QoS policy for a service provider hosting webmail portals for two different companies.

site are needed and no instrumentation is required. In other words, to make an Internet site capable of providing QoS guarantees it is enough to deploy Quorum at its entrance point and define the desired QoS policy to be enforced.

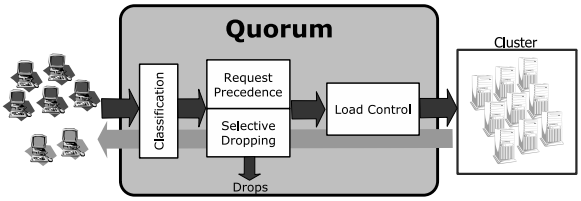


Figure 2: The architecture of Quorum.

Figure 2 depicts the architecture of Quorum, consisting of four different modules each of which implements part of the functionality that is necessary to enforce a QoS policy. The *Classification* module categorizes the intercepted requests from the clients into one of the service classes defined in the QoS class. The *Load Control* module determines the pace (for the entire system and all client request streams) at which Quorum releases requests into the cluster. The *Request Precedence* module dictates the proportions with which requests of different classes are released to the cluster. The *Selective Dropping* module drops requests of a service class to avoid introducing work accumulation that would cause a QoS violation. In the next sections we detail further the implementation of the Quorum modules. We explicitly exclude the details associated with Classification since it is a well understood problem that has already been studied in the literature [22].

3.1 Load Control

The Load Control module has two primary functions. First, it prevents large amounts of incoming traffic from overloading the internal resources of the cluster. When the internal resources become overloaded, the internal software (i.e., operating system, web servers, applications, etc.) will delay or drop requests without regard for their QoS classification. Second, it maintains the resources within the cluster at a high level of utilization to achieve an overall good system performance (for the given cluster configuration). The goal of the Load Control module is to have the cluster operate at maximum

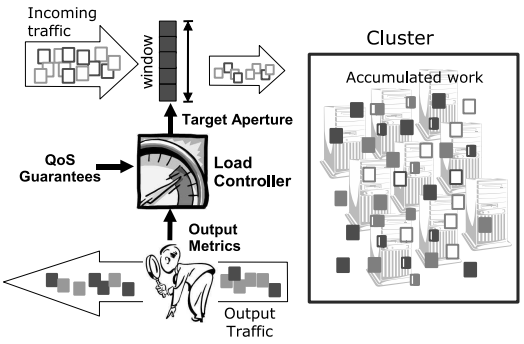


Figure 4: Structure of Load Control module.

capacity so that the largest possible capacity guarantees can be met, while also preventing overload conditions that would cause response time guarantees to be violated.

Based on the previously described time-shared model, our implementation exploits the direct correlation between the amount of work accumulation inside the cluster and the time required for requests to be computed by the hosted services. In general, more work introduced into the cluster corresponds to longer compute times for each service (given a fixed amount of resources) once the number of parallel requests exceeds the number of resources. With this in mind, the Load Control module can directly affect the amount of time that requests take to be computed inside the cluster (i.e., compute time) by controlling how much traffic is "in progress" at any time.

Similar to TCP, our implementation uses a sliding window scheme that defines the maximum number of requests that can be outstanding at any time (see Figure 4). The basic operation of the Quorum engine consists of successively incrementing the size of the window until the compute times of the QoS class with the most restrictive response times approaches the limits defined by its guarantees. Our current implementation uses a simple algorithm (see Figure 3.a) that increments (or decrements) the window linearly until the currently observed computing times are considered too high by the Selective Dropping module (see Section 3.3 for details on how this threshold is chosen). Our implementation updates the window size every 500ms, a compromise between having fast reaction times and allowing enough requests to finish within a period such that the collected computing times are significant. A more sophisticated (and reactive) version of the algorithm using non-linear variation of the window sizes is under study.

3.2 Request Precedence

The main function of Request Precedence is to virtually partition the cluster resources among each of the service classes. Resource isolation is a necessary functional-

```
Load Control Algorithm

recalculateWindow()  // every Δ milliseconds
   CT(t) =  Observed Compute Times (t,t-Δ)
   RTG_min = Strictest Response Time Guarantee
   if CT(t) <  RTG_min/2 then
      window++  // linear increase
   else
      window--   // linear decrease
```

(a)

```
Request Precedence Scheduler

getNextRequest()    (minUsage = ∞)
   foreach class C
      //compute its weighted instant usage
      usage = C.outstanding/C.weight
      //identify class with least usage
      if  usage<minUsage and ! C.isEmpty()
            targetClass = C ; minUsage = usage
   return targetClass.dequeueRequest()
```

(b)

```
Selective Dropping Algorithm

isDroppable(request)
   CT(t) =  Observed Compute Times (t,t-Δ)
   RT_max = Response Time Guarantee
   expectedRT = request.waitingTime + CT(t)
   if expectedRT > RT_max then
      return true  // Drop
   else
      return false // Forward
```

(c)

Figure 3: Simplified pseudo-code for the algorithms of the three main modules of Quorum.

ity that allows each service class to enjoy a minimum amount of processing capacity, independent of potential overload or misbehavior of others. This module is able to partition externally the service delivered by the cluster, by controlling the proportions in which the input traffic for each class is forwarded to the internal resources. Thus, the goal of this module is to ensure that the fraction of the overall cluster capacity devoted to each class is large enough to satisfy its throughput guarantees at all times.

The Request Precedence module also attempts to maximize performance in overload situations without allowing guarantees to lapse. It reassigns unclaimed resources to other QoS classes demanding more processing power than they have been granted. Reassigning unutilized capacity allows the QoS engine to take full advantage of the available cluster resources allowing some service classes to enjoy a level of service that is higher than what they have been guaranteed. At the same time, the Request Precedence module ensures that those classes that are not using their maximum allowable share of the overall capacity none-the-less receive enough capacity to meet their guarantees. By continually calculating and adjusting the fraction of cluster capacity that is given to each class, Quorum differs from an approach that relies on physical partitioning of the resources where temporary reassignment cannot be implemented.

Under Quorum, Request Precedence is implemented by a scheduling algorithm that logically partitions the window of outstanding requests (as dictated by the Load Control module) according to the throughput guarantees specified in the QoS classes. This method exploits (and depends on) the time-shared nature of clusters which assign resources equally amongst all running tasks. As a result, it is possible to increase the share of the cluster resources for a particular service class by increasing the number of tasks that are devoted to computing its requests. In particular, our model defines that each request that is being computed in the cluster will get $\frac{1}{N}th$ of the total utilization, when N is the number of requests that are concurrently being processed in the cluster. Therefore, the aggregated utilization of M requests will amount to $\frac{M}{N}th$ of the total cluster usage. In the case where all of the M requests belong to the same service class, that service class is effectively enjoying a $\frac{M}{N}th$ of the cluster's resources.

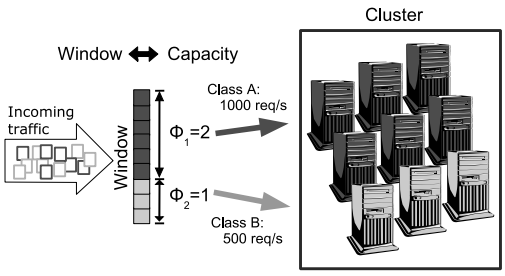


Figure 5: Function of Request Precedence module.

To that effect, our scheduler assigns a weight Φ_i to each of the QoS classes and uses this weight to partition proportionally the window accordingly (see Figure 5). Instead of allocating a fixed number of slots of the window per class, our algorithm (see Figure 3.b) uses a dynamic method that achieves similar characteristics to Weighted Fair Queuing disciplines [11, 17, 31] in terms of proportional rate guarantees and reassignment of surplus. However, the guarantees in our case apply to window sizes instead of service rates (i.e., throughput). The reason for this choice is that throughput for a given service class can only be guaranteed when the computing requirements of the requests are known. In other words, the capacity necessary to achieve a given throughput is directly related to the computational complexity of the requests. On the other hand, assigning a particular window size corresponds to guaranteeing a portion of the cluster capacity, independent of the computing complexity of the incoming request stream. Therefore, by working with a capacity measure (i.e., proportions of outstanding requests), Quorum can provide effective isolation between classes when their computing requirements are not known *a priori* or can change dramatically.

Capacity can be seen as a fungible metric that links output throughput and computing requirements such that an increase in one can be made to force a decrease in the

other. For example, a capacity equivalent to 10 nodes may correspond to an output throughput of 500 req/s at a compute cost of 20ms/req, but also to 1000 req/s if the compute cost is only 10ms/req. The internal capacity allocated for a class is calculated from the *nominal* guaranteed throughput (as expressed in the QoS class) and the *expected* computation requirements of the requests (as agreed upon between the provider and the consumer). In the cases where the computation complexity is violated (i.e., higher than agreed upon) for a particular class, instead of dropping the traffic of the faulty class, Quorum will gracefully degrade its throughput to maintain the same internal capacity allocation.

3.3 Selective Dropping

The function of Selective Dropping is to discard the excessive traffic received for a QoS class in the situations where there is not enough available capacity to fulfill its incoming demands. A dropping module is necessary to prevent large delays from occurring in overloaded situations where requests would otherwise accumulate unboundedly in the engine and violate the QoS guarantees. The goal of the Selective Dropping module is to ensure that the response time guarantees of each class will be met for all requests that can be serviced.

The basic operation of the Selective Dropping module is, in essence, very simple, since it can leverage from properties that are already provided by the Load Control and Request Precedence modules. The Selective Dropping module independently observes each of the QoS queues of the engine and discards the requests that have been sitting in the queue for so long that the deadline for their service cannot be met. In our implementation (Figure 3.c), a request will be dropped if the time left for meeting the deadline once it gets at the head of the queue is less than the expected time of computation of its class. In other words, a request will be dropped if we expect it to miss its deadline according to how other requests of the same class are currently performing (Figure 6). The current computation times for a class can be considered a reliable estimation of their expected computation since they are stabilized by the feedback loop of the the Load Control module.

The Selective Dropping module leverages the queuing inside Quorum to absorb safely peaks of traffic during transient overload conditions without violating the response time guarantees. To this effect, it works closely with the Load Control module by signaling ahead of time when a service class is likely to become overloaded. In our implementation we signal the Load Control module to stop increasing the load of the cluster when the observed computing time of the most restrictive service class reaches half of its response time guarantee. By

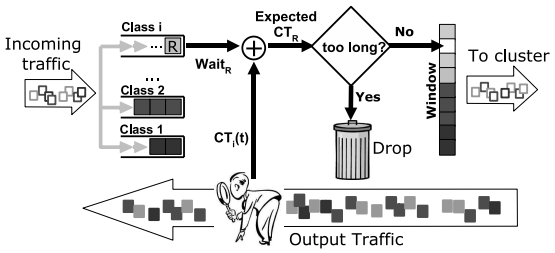


Figure 6: Structure of Selective Dropping module.

closely working with the Load Control module, Selective Dropping can ensure that there is an available queuing time that is at least half the maximum allowed response time. Note, however, the queuing time is independent for each class, therefore classes with looser response time guarantees can support longer queuing periods and thus absorb of much larger transient peaks of traffic without violating the guarantees. The choice of 'half' is a compromise motivated by the tradeoff between maintaining cluster occupancy and allowing the necessary queuing space to absorb peaks of traffic. We are currently working on an optimized version that can dynamically adapt this threshold to allow more queuing without adversely affecting overall system performance.

Finally, the Selective Dropping module must also ensure that dropping requests of a particular service class does not incur in a violation of its throughput guarantees. To ensure that there are no invalid drops, our module relies on an important property of the Request Precedence module which states that the forwarding rate of requests for a class into the cluster will be no lower than its guaranteed throughput. This property is derived from the capacity guarantees and the windowing system of the Request Precedence module which allows a new request to be forwarded immediately after one of the same class finishes. Therefore, this property ensures that requests will only accumulate in the engine if the incoming rate of a class surpasses its guaranteed throughput, in which case drops can safely be executed since they would not violate any throughput guarantees. Note that these properties do not hold true for misbehaving classes where the computation requirements of incoming requests are higher than expected. However, this is not a problem since we have already discussed that QoS guarantees do not need to be met for such classes, which can be penalized both in terms of throughput and response times. The implementation of independent dropping techniques, coupled with the guarantees given by Load control and Request Precedence, allow this module to provide response time guarantees and isolate one class against misbehavior of others.

Combined, the functions of all four Quorum modules (Classification, Load Control, Request Precedence and Selective Dropping) enable cluster responsiveness, efficient resource utilization, capacity isolation and delay differentiation, thus guaranteeing capacity and response times for each independent service class.

4 Experimental Performance Comparison

In this section we demonstrate that the four modules of Quorum can provide QoS guarantees under realistic conditions even though they treat the cluster resources and Internet services as a "black-box". We have performed extensive studies of each of the presented modules, both in isolation as well as operating together. Due to space constraints we do not include them in this paper, but the details of these studies can be found in [12]. Instead, in this section we focus on examining the performance of Quorum as a complete system, and study how it compares to the best of the known approaches. Our investigation is empirical and is based on the deployment of an Internet search service used by *Teoma* [35] using a 68-CPU cluster. We analyze how five different techniques (representing both state-of-the-practice and state-of-the-art) offer differentiated quality to distinct groups of customers using generated message traffic based on web-search traces. We then quantify the observed quality of service delivered by each method.

4.1 Experimental Methodology

Our experimental setup consists of several client machines accessing a cluster system through an intermediate gateway/load-balancer machine. Accessing the services through a load balancer machine is the most commonly used architecture in current Internet services. For example, Google [21] funnels traffic through several Netscaler [29] load-balancing systems to balance the search load presented to each of its internal web servers [23].

To perform our experiments in the most realistic possible manner, we have deployed a commercial-grade Internet service on a 68-CPU cluster system and replayed real traffic traces from its commercial operation [33]. The service deployed is the index search component of the Teoma commercial search service [35]. The index search component consists of traversing an index database and retrieving the list of URLs that contain the set of words specified in the search query. The total size of the index database used is 12GB and is fully replicated at each node. The index search application from Teoma is specifically built for the Neptune middleware [34], a cluster-based software infrastructure that provides replication, aggregation and load balancing for

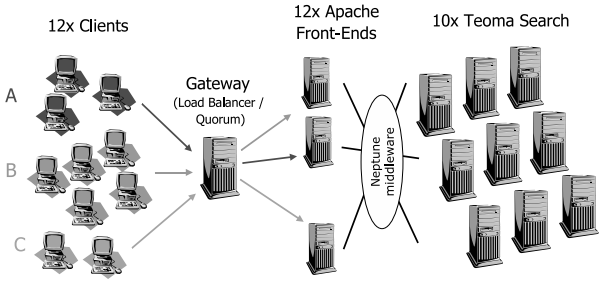


Figure 7: Experimental test-bed used for our benchmark using Teoma's search service.

network-based services. The version of Neptune we use also provides QoS mechanisms allowing the specification of proportional throughput guarantees and response times constraints through the definition of yield functions [33]. As it is the case with commercial search engines, our system accesses the service through a set of front-end machines that transform the received URLs into internal queries that are then forwarded to the middleware servicing the search database for processing. To mimic the environment at Teoma, we implement the front-end with an Apache web server [3] and a custom-built Apache module that interfaces with the Neptune infrastructure. This module is necessary to utilize the middleware functionality to locate other Neptune-enabled nodes and appropriately balance the requests based on the current load of the available servers. The cluster configuration used in our experiments is depicted in Figure 7. The hardware configuration of the cluster consists of 2.6 MHz Intel Xeon processors each with 3 gigabytes of main memory organized into nodes with either two or four processors per node. The network interconnect between processors is switched gigabit Ethernet and the host operating system is RedHat Linux/ Fedora Core release 1, using kernel version 2.4.24.

Our gateway node is a 4-CPU dedicated machine that can function in two different modes: as a load-balancer or as the Quorum engine. When running in load-balancer mode, the machine is configured to implement the typical (Weighted) Round Robin and maximum connections options available in most commercial hardware [19, 20, 29]. When running as Quorum engine, the gateway is configured to enforce the QoS policy defined for the experiment. Both the load-balancer and Quorum engine are entirely implemented in user-level software. The gateway is implemented as an event-driven Java application which makes extensive use of the new libraries for improved I/O performance [24]. We use Sun's 1.5 Java virtual machine with low-latency garbage collection settings. Our performance tests show that our implementation can achieve a peak performance of 12Kreq/s (i.e., around 70K packets/sec) for certain client work-

QoS Class	Classification	Output Guarantees	
	Layer 7 pattern	Throughput (Avg. req/s)	Response Time (Avg. ms)
A	Host: A	375	200
B	Host: B	937	600
C	Host: C	562	300

Experimental Workload	
Input (Avg. req/s)	Service Status
185	Not Overloaded
1718	Overloaded
557	Fully Utilized

Table 2: QoS guarantees and traffic workload of the Teoma search engine benchmark.

Throughput (req/s)							
Class	Input	Guarantee	Load Balancer	Physical Partitioning	Overprovi-sioning	Neptune QoS	Quorum
A	185	375	148 (8)	185 (9)	185 (10)	185 (9)	185 (10)
B	1718	937	1333 (33)	986 (41)	1718 (130)	1133 (68)	1059 (68)
C	557	562	443 (16)	557 (26)	557 (29)	548 (23)	557 (27)

Response Times (ms)						
Class	Guarantee	Load Balancer	Physical Partitioning	Overprovi-sioning	Neptune QoS	Quorum
A	200	13335 (3175)	40 (43)	40 (22)	45 (50)	121 (46)
B	600	13395 (3201)	18937 (3734)	75 (116)	256 (180)	600 (37)
C	300	13487 (3106)	305 (218)	19 (19)	62 (49)	134 (53)

Table 3: Experimental results for Teoma search engine.

loads. Thus the performance of our base-level system is high enough to be used in load levels that are comparable to current commercial systems (e.g. Google reports around 2500 req/sec [23], Ask Jeeves around 1000 req/sec [5]). Both our implementation of a load-balancer and the Quorum engine are based on the same core software for fielding and forwarding HTTP requests.

For this experiment our methodology consists of using the previously described test-bed to recreate search traffic and to explore the effectiveness with which five different approaches can enforce a particular QoS policy for a single service with multiple client groups. The five compared approaches are:

Load Balancer The gateway machine is configured as a load balancer and tuned to match common high performance settings of Internet sites. Specifically, we configure it to use the least connections load-balancing algorithm and limit the maximum number of open connections for each front-end to match their configured maximum (i.e., 250 processes for Apache server and 150 for the Tomcat engine).

Physical Partitioning A separate group of machines are dedicated for each of the existing QoS classes. We configure the load-balancer to forward requests of a particular class only to its restricted set of reserved nodes.

Overprovisioning The size of each physical partition is increased such that the resulting capacity and response time guarantees can be achieved as specified by the QoS policy (possibly at the expense of under utilized resources).

Neptune QoS The gateway is configured as a load balancer and the QoS mechanisms of Neptune are enabled to implement the QoS policy under study.

Quorum QoS The gateway runs the Quorum engine which implements QoS and the internal cluster resources implement only the Internet service. (i.e., QoS functionality in Neptune is disabled)

In order to benchmark Quorum and the other considered QoS methodologies, client requests are replayed from a request trace supplied by Teoma that spans 3 different days of commercial operation [33]. We also use Teoma-supplied traces of word sequences to generate real search queries. The levels of incoming traffic are designed so that the input demands of the different clients are far below (class A), far above (class B) and coinciding with (class C) the capacity constraints specified in their respective QoS classes. Clients for each QoS class use different inter-arrival times, corresponding to one of the three different days of the original traces. Table 2 further depicts the details of the QoS policy and input workload used in the experiment, including the capacity and response time guarantees for each QoS class.

4.2 QoS Results

Figure 8 presents the results in terms of achieved average throughput and average response times for the five QoS methodologies using the same input request streams. The upper portion of the figure shows how the totality of incoming traffic for a class (represented by the height of a bar) has been divided into traffic that is served and traffic that is dropped. Horizontal marks delimit the minimum amount of traffic that has to be served if the QoS guarantees are met. Note that a resulting throughput below the horizontal marks still meets the QoS guarantee for a class if the totality of its incoming traffic is successfully served (i.e., the system cannot serve more traffic than it has received). The lower part of Figure 8 presents the results in terms of response times. For response times, we use horizontal marks to denote the maximum response times allowed by the QoS policy and denote with a darker color the classes that do not meet the guarantees. We present these response time results using a logarithmic scale for better visual comparison since the delays differ substantially. Table 3 summarizes these results in tabular form (including standard deviations in parenthesis) to further aid their comparison.

We begin by analyzing the quality of the service achieved by a load-balancer-only technique. Throughput results show that the amounts of traffic served in this case are directly dependent on the levels of incoming traffic

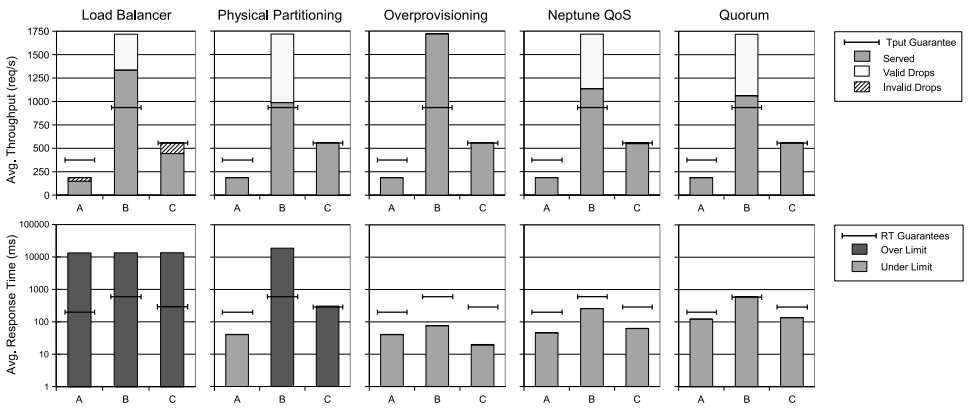


Figure 8: Experimental comparison of current approaches and Quorum, using Teoma's search engine.

rather than driven by the specified QoS policy, thus isolation between classes is not achieved. In this case we see that the dominance of class B traffic induces drops in A and C, even though the demands for these classes are always below (in the case of class A) or never exceed (for class C) the guaranteed capacity for each class. At the same time, the large response times shown in the lower figure, demonstrate that simple connection limiting techniques employed by the load-balancer are not enough to prevent large delays in response times (e.g. up to 14 seconds per request), rendering this technique inadequate to provide QoS guarantees.

When resources are physically dedicated through Physical Partitioning, the system is able to serve the expected amount of traffic for each of the classes and drop requests only in the cases when the demands of incoming traffic exceed the allocated capacity. Throughput guarantees are met, however, if we observe the results in terms of response time, we see that the overloaded partition B experiences a delay more that 30 times higher than the maximum allowed by the QoS policy. Thus while physically partitioning resources is able to provide capacity guarantees, it fails to ensure response times constraints for arbitrary incoming demands. It is worth noting that the reason for partition B serving more throughput than its guarantee is that the raw performance of the partition is slightly higher than the QoS guarantee defined in the policy.

When each of the partitions is augmented with enough resources (i.e., over-provisioning) all requests are successfully served. The response times are also reduced below the maximum allowed delay. In this case, class B and class C require an additional 10 and 2 CPUs respectively in order to meet the specified response time guarantees. Thus over-provisioning is the first of the techniques that can successfully provide both throughput and response time guarantees. However, meeting the QoS guarantees through over-provisioning comes with a high cost. In our experiment, the increase in cost of overprovisioning was 60% (i.e., from 20 to 32 CPUs) with resource utilization declining to 80%. Further, these numbers represent the *minimum* amount of over-provisioning that allowed us to achieve the QoS goals. In general, between load spikes the extra resources needed to serve surges in load lay idle. Thus, given the wide load fluctuations that most commercial Internet services can experience (i.e., 3-10 times the normal amount [16]), we expect the resource utilization of over-provisioned systems *in situ* to be worse than what we observe in this experiment.

Neptune QoS and Quorum both meet the specified throughput and response time guarantees. Both techniques serve at least the necessary amount of traffic and are able to keep response time below the maximum delays associated with each guarantee. Furthermore, both techniques are able to successfully reassign the capacity not utilized by class A to the greedy clients of class B. We observe that direct control of the resources and services in the cluster (due to its invasiveness) allows Neptune to achieve a slightly better throughput than Quorum (i.e., 3%). This slight performance penalty can be seen as the cost that an external solution such as Quorum has to pay for not modifying any of the software internals. However, given the completely non-invasive nature of Quorum, we were surprised by how closely it matched the performance achieved by the invasive and commercially developed Neptune system. Figure 8 also shows that the resulting response times from Neptune are somewhat lower than Quorum. This difference is because Quorum is only designed to enforce maximum delay constraints and it is not concerned about minimizing the overall delay of service times. We are currently working on a prototype that can both ensure response time constraints and lower response delays when possible.

Summarizing, this experiment demonstrates the effectiveness of Quorum empirically, using a commercial Internet service and commercial traffic levels. Quorum in

this setting is competitive with the best of the current approaches in its ability to enforce both response time and throughput QoS guarantees. In particular, Quorum has less cost and achieves better resource utilization than over-provisioning techniques due to its ability to reassign unutilized capacity to those service classes that need it. At the same time, it achieves comparable QoS guarantees to an integrated and commercially available system such as Neptune, incurring only a small performance cost (i.e., 3%). In the next section (Section 5.4) we illustrate its flexibility by showing how it can provide reliable QoS guarantees in a complex and heterogeneous site running three different services.

5 Robustness under Extreme Conditions

In this section we investigate the robustness of Quorum and its QoS enforcement capabilities under scenarios that emulate the extreme conditions experienced by many current Internet services. To do so, we first study the reaction of Quorum to three circumstances: sudden traffic fluctuations (Section 5.1), sudden changes in computing requirements (Section 5.2) and node failures and recoveries (Section 5.3). We then present a larger-scale experiment in which we detail its response to the same conditions in a substantially more complex Internet hosting scenario (Section 5.4).

To conduct the initial set of isolated robustness studies we use two service classes: A and B. Service class A is a misbehaving class that begins with an input load that can be fully serviced with its allocated capacity, and then changes its demands to surpass the capacity required to meet its guarantees as well as to drive the overall system into overload. Service class B is a well-behaved class that receives a constant demand of traffic that is always below the traffic level that can be serviced under its guarantees. For each of the experiments, we detail how well Quorum insulates the quality of service experienced by the well-behaved class B from the fluctuations introduced by class A. We also investigate how the quality of service given to class A degrades gracefully during the periods when its demands exceed the capacity allocated to meet its guarantees. In particular, our goal is to provide as much capacity to A as possible without violating the guarantees made to either A or B. As described in subsection 3.2, however, the capacity allocated to A and B is fungible and constantly adjusted by Quorum as it responds to changes in load conditions.

To run these experiments we use a system consisting of 4-CPUs for client machines accessing a 16-CPU cluster through a gateway machine implementing the Quorum engine. Each of the servers runs the Tomcat application server [36], providing a "CPU-loop service" consisting of a servlet that loops a number of times so that it utilizes a certain amount of CPU (as specified in the HTTP parameters of each incoming request). This artificial emulation of a true web service allows precise control of the CPU load requirements associated with each request. Requests received from the clients are classified into QoS classes according to the host field name found in the HTTP header of the request (i.e., host: A or B).

QoS Class	Classification	Output Guarantees	
	Layer 7 pattern	Throughput (Avg. req/s)	Response Time (95^{th}% ms)
A	Host: A	900	400
B	Host: B	900	400

Table 4: QoS policy used in the studies.

The QoS policy defined for the experiments allocates the same guarantees for both classes of service (Table 4). Note that unlike the previous experiments, the response time guarantees are expressed in terms of 95th percentiles and not averages – a much more challenging but potentially more desirable metric to enforce, especially given the range of conditions to which we subject the cluster. All figures in this section depict the resulting average of the observed throughput (upper graph) and the 95th percentile of response times (lower graph) over two-second sampling intervals.

5.1 Sudden Traffic Fluctuations

In this experiment we show how Quorum manages wide fluctuations of incoming traffic. To demonstrate this property we subject the service for class A to a sudden-but-sustained impulse of incoming traffic that is four times its normal rate. This sudden increase in demand is enough to bring the cluster to full utilization. Figure 9 shows the results from the experiment. In the Figure, the traffic fluctuation (labeled as "Input Class A") increases instantly from 600 req/s to 2400 req/s 120 seconds after the experiment has begun. Despite the sudden and sustained increase in A's traffic the degree to which service class B meets its guarantees is isolated from the change in input conditions. B's throughput is virtually unaffected and its response times, while they climb, are always kept below the maximum guaranteed delay. In response to the traffic surge, Quorum quickly shifts any uncommitted resources to class A. Strictly speaking, it is consistent with the guarantee given to class A simply to cap throughput at 900 req/s for that class. However, by automatically sensing the degree to which it can slow down B's response times (without violating B's guarantees) and committing additional resources to A, Quorum is able to give A as much throughput as can be spared while remaining within the constraints of both guarantees.

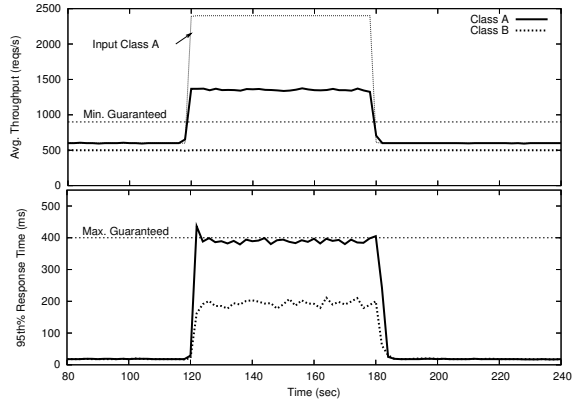


Figure 9: Quorum's reaction to extreme fluctuation of incoming traffic.

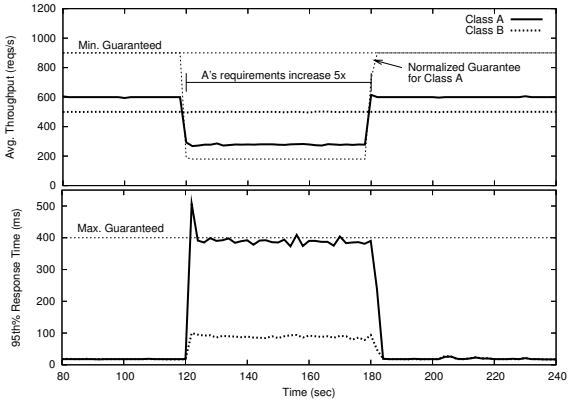


Figure 10: Behavior of Quorum when requests of class A suddenly require five times more resources for their computation.

We should note that the slight spike in response times occurring in second 120 appears a consequence of our short sampling period. We wish to depict circumstances that stress the capabilities of Quorum and as such, we calculate the percentiles with a two-second periodicity. In practice, it is unlikely that a commercial system will need to ensure QoS guarantees on such a fine-grained time scale, especially when using percentiles to specify guaranteed performance levels.

5.2 Computing Requirements Overload

In this experiment we investigate how Quorum handles wide variations in the computing requirements associated with a request stream. These types of variations can occur in situations such as application misbehavior (e.g., software bugs that cause excessive resources to be used in computing a request) or changes in the workload characteristics (e.g., requests incurring in unusually long and expensive database queries). We induce this anomaly by suddenly increasing the computing requirements for class A from 8ms to 40ms of exclusive CPU time. Again, the goal is to protect the performance of class B while degrading the throughput given to class A to a level that is both maximal and consistent with the guarantees for both classes. To better observe the expected service for class A we include the throughput guarantees normalized to its incoming computing requirements (i.e., the normalized throughput is five times lower than the nominal when requests are five times more difficult to compute).

Results from the experiment are depicted in Figure 10. As in the previous experiment the throughput given to class B remains virtually unaffected by the increase in computing requirements (seconds 120-180), and its response times are always kept below the guarantees. At the same time, in response to the increase in computing demands for the misbehaving class A, Quorum immediately decreases A's throughput. Although degraded, A's throughput is always maintained above the normalized guarantee corresponding to the internal capacity allocation Quorum made for this guarantee.

Recall from Section 3.2 that the Request Precedence module guarantees enough resources to class A to fulfill the nominal throughput guarantee of 900 req/s assuming 8ms of computing time. When the computing requirements increase to 40ms/req the throughput must be lowered to 180 req/s to preserve enough capacity for B's guarantees. Thus we expect the system to enforce a throughput guarantee of 180 req/s for class A during the period in which its requests require 40ms of CPU time, as shown by the normalized guarantee line. However, between seconds 120 and 180 of the experimental period, class A is receiving a throughput of 280 req/s, which includes a surplus of 100 req/s corresponding to the resources that class B is not utilizing. If B's requirements were to suddenly increase, Quorum would reduce A's throughput to 180 req/s and and change the proportion of B's requests admitted to reallocate more resources to B. Note also that this constant allocation and reallocation of capacity is sensed by the Quorum engine automatically based on the observed responses leaving the cluster, and not based on predefined parameters or instrumentation describing the CPU requirements for each type of request. As is the case with the previous experiment, the short time scale over which each percentile is computed causes a single "spike" in response time during the two-second interval spanning second 120.

5.3 Node Failures and Recoveries

In this experiment, we depict Quorum's response to significant node failures and recoveries. At second 120, we induce the failure of 2 out of the 8 nodes and then recover

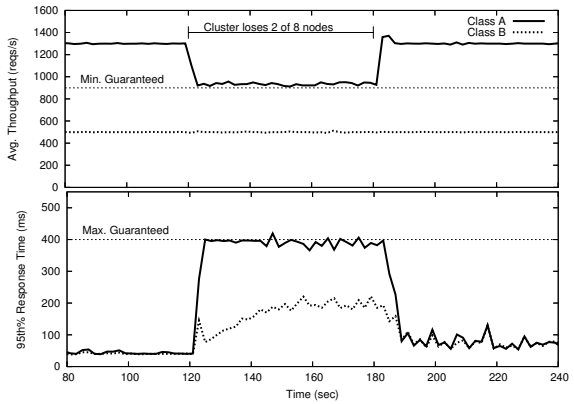


Figure 11: Quorum's reaction to a failure of 2 nodes.

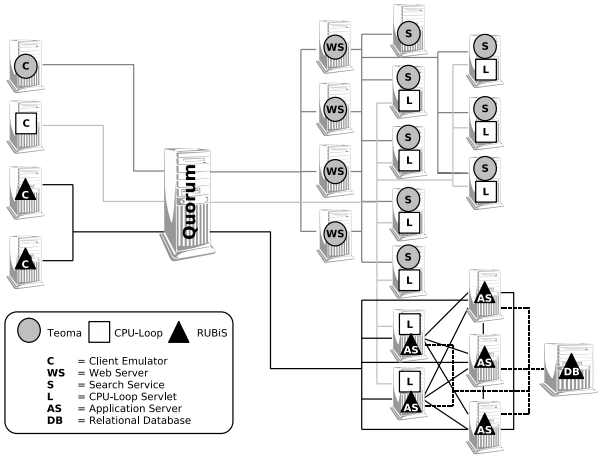


Figure 12: Setup of the complex, heterogeneous Internet site.

QoS Class	Classification	Output Guarantees	
	Layer 7 pattern	Throughput (Avg. req/s)	Response Time (95^{th}% ms)
Teoma	Port=8888	450	600
CPU-Loop	Port=9999	1350	3000
RUBiS	Port=10000	700	3500

Table 5: QoS policy for the complex and heterogeneous Internet site.

the nodes 60 seconds later. To introduce these failures we program our load-balancer module to stop forwarding traffic to the "failed" nodes. We have also increased the incoming traffic rate for class A to 1300 req/s in order to make the resulting change in throughput more visible.

We show the results of the experiment in Figure 11. When the nodes fail, Quorum rapidly reduces the throughput given to class A to its 900 req/s guarantee. Notice that this adjustment, again, does not violate the quality of the service guarantees given to class B. As with the previous two experiments, the throughput for B is unaffected while the response times grow to a level well below their maximum guaranteed delay.

We should note that in this example it was possible to enforce the QoS policy, even under the degraded operation, because there was enough spare capacity that B was not utilizing which could successfully be reassigned to A. In the cases where there are not enough resources to fulfill the guarantees across all classes (i.e., QoS policy is not feasible), Quorum reacts by degrading the service of each class proportionally to the guarantee associated with that class. For example, if the input demands for class B had been above the guaranteed 900 req/s, the Quorum would have evenly assigned a throughput of 700 req/s for each class since the degraded capacity of the system would support 1400 req/sec in total, and the guarantees for both A and B are the same. We believe that other non-proportional mechanisms for reapportioning fungible capacity when QoS policies become infeasible are highly desirable and we plan to investigate them further in our future work.

5.4 Complex Heterogenous Services

Through the previous set of controlled experiments we have shown that Quorum can both enforce service isolation as well as gracefully degrade the service of misbehaving classes even under extreme operating conditions. We now show how Quorum reacts to the same three severe circumstances for a larger-scale and substantially more complex Internet site that hosts three different services. Additionally, this experiment illustrates the flexibility of Quorum's "black-box" approach: its ability to provide QoS guarantees using heterogeneous hardware configurations and multi-tiered software architectures where the source code of the applications cannot be modified. At present, we know of no other published infrastructure that can provide QoS for this complex Internet hosting scenario.

To perform this experiment we host the Teoma search and CPU-loop services (described previously) together with a third service called RUBiS [32] using shared set of cluster resources. RUBiS is a publicly available auction site modeled after eBay that has been used by several researchers for evaluating application server performance scalability [14, 15]. We use the version of RUBiS that is implemented using Enterprise Java Beans (EJB) deployed on top of JOnAS application server (v3.3.6) and Tomcat (v4.1) servlet engine. The Tomcat servers are configured with session replication and the JOnAS application server is configured to balance the execution of EJBs across each of its nodes according to their respective loads. The auction data is stored using a mySQL database with the same configuration and size as the benchmark described in [15]. Traffic for the RUBiS auction is generated by the client emulator supplied with

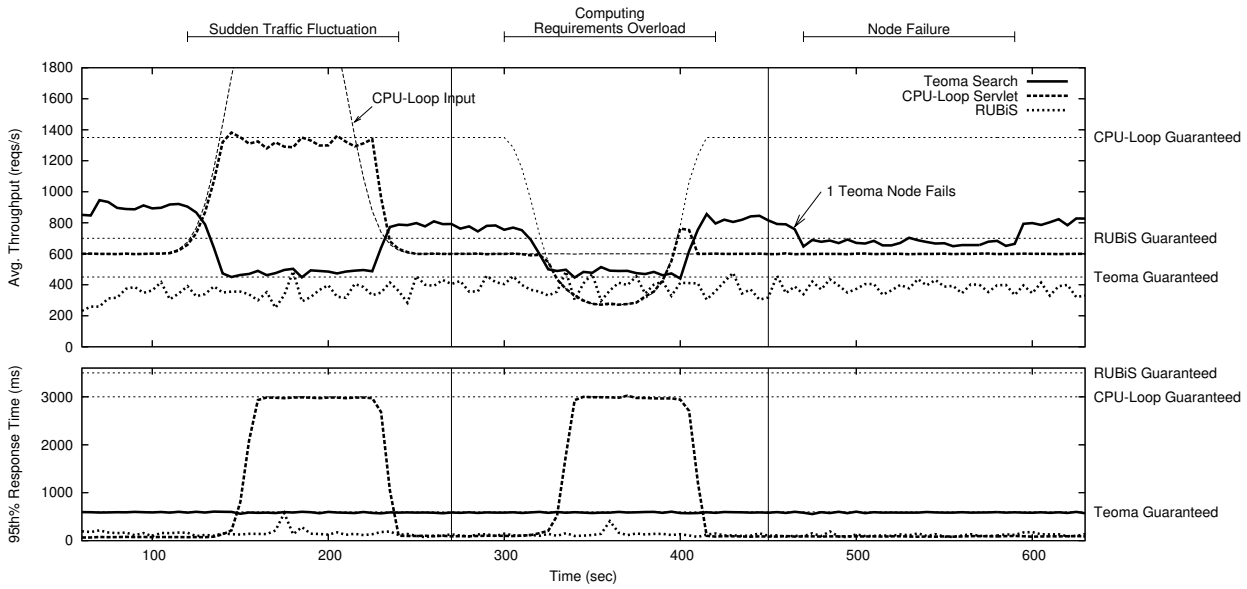


Figure 13: QoS results for a complex, heterogeneous Internet site.

the RUBiS software which performs typical actions of an auction user such as browsing, bidding or buying items. This type of service also allows us to illustrate the efficiency of Quorum when providing QoS guarantees under highly variable workloads. In this particular case the distribution of computation requirements resulted in (*median=101*, *mean=149*, 95^{th}*%=457*, *max=3088*)ms which can be approximated with a two-phase hyper-exponential with the first mode on the mean.

Figure 12 depicts the hardware and software configuration used for this experiment. Notice that we include both nodes that are dedicated to a single service as well as nodes that are shared by more than one service. In particular, the CPU-loop service shares 7 of the 8 nodes used by the Search component of Teoma, and also with 2 of the 5 nodes running the RUBiS auction. Our intention is to capture both the fluid sharing of cluster resources as well as the static capacity planning that we believe will always be present in a commercial system.

Also for this experiment we program our Quorum engine with the QoS policy defined in Table 5, deploy it at the entrance of the site (with no other information than the QoS policy), and observe how well it performs in response to the same three types of changes explored in the previous subsections. Similarly, we generate three types of input load. For the Teoma service, we introduce incoming traffic that exceeds what can be completely serviced under the constraints of its guarantee. Alternatively, for the RUBiS service, we keep the incoming traffic load below the maximum serviceable level. We then vary the input for the CPU-loop service to create a peak of demand during the period from seconds 140 to 220 and to increase its computing requirements from 8ms to 40ms during the period between seconds 300 and 420. Finally we kill one of the Teoma back-end nodes at second 475 and restart it 120 seconds later.

Figure 13 shows the evolution of throughputs (above) and response times (below) for each of the three different services during the 11 minute run, in which a total of 1.1 million requests were served. Vertical lines separate the three different conditions (input increase, computation increase, node failure) to which Quorum must respond. Throughput guarantees are again normalized to the expected computing requirements. Only CPU-loop service shows a deviation form the nominal throughput guarantees since it is the only service that suffers a change in its computation requirements. From the first segment of the figure, it is evident that Quorum protects the RUBiS service and also reassigns the the available resources such that the two overloaded classes during the peak period are served according to the QoS policy. As we observed in Section 5.1, the amount of surplus service received by Teoma during the peak period, is given back to the CPU-loop service so that both classes can operate at their limits of throughput and response times.

In the second segment of the figure, the computing requirements of CPU-loop service increase to 5 times their original levels. In this case we induce a change in the computing requirements that it is more gradual than the sharp change shown in Section 5.2 to better emulate how a true Internet site might degrade. Quorum reassigns capacity not needed to meet Teoma's guarantees to the

CPU-loop service while maintaining the guarantees for RUBiS. Also, the CPU-loop service suffers a degradation in throughput that is inversely proportional to the increase in its computing requirements, thus maintaining the fungible capacity described by its guarantee. In this case, there are no extra resources to be used in aiding the overloaded CPU-loop class, thus its resulting throughput is capped exactly at its normalized guarantee.

In the third segment of the experiment the dedicated search back-end from the Teoma service fails. In this case we induce a true failure by killing the server process of Neptune and use the fail-over and recovery capabilities of the middleware to detect the change. Note that the failure of the node only has an effect in reducing the spare capacity that Teoma service is enjoying. Both the throughput and response times of CPU-loop and RUBiS are, once more, unaffected.

The results from these experiments illustrate several important points. First, our prototype implementation of Quorum is able to provide *robust* QoS guarantees even in the presence of the extreme conditions which service providers are currently facing. Second, the QoS guarantees are provided in very *fine-grained* time scales even when using a strict metric such as the 95th percentile of the response times. Third, Quorum is a *flexible* QoS solution that can provide performance guarantees in heterogeneous Internet sites without requiring any prior knowledge of their internal hardware architecture or software configuration. Forth, Quorum is capable of handling *complex service types* which can exhibit wide (legitimate) variations in the computation requirements of their requests (e.g., RUBiS auction). In summary, our empirical evaluation shows that Quorum is a viable solution to QoS provisioning for Internet services, that has the robustness and flexibility that current service providers seek without requiring the modification of any of the existing software infrastructure of the sites.

6 Related work

There are many approaches to providing QoS for Internet services, but relatively few that combine flexibility and extensibility with response time and throughput performance. In this section we briefly introduce some of the most relevant work and compare it to the Quorum approach.

QoS for network communication is typically defined in terms of reliable communication between two endpoints with performance guarantees. Protocols such as diffserv [9] and intserv [13] or trunk reservation schemes [28] leverage the existing routing infrastructure and network knowledge to provide bandwidth allocation and packet delay guarantees over the Internet. At a higher level, approaches such as Content Distribution Networks [1] provide similar features by appropriately managing an overlay network to content closer to the end-user. These approaches focus on the communication component and do not address the computational requirements associated with the servicing of Internet requests. In contrast Quorum works at the boundary of the cluster hosting the services and, as such, complements approaches that ensure quality of network service between the client and the cluster.

Load balancers [19, 20, 29] are perhaps one of the the most closely related approaches to Quorum. Properly tuned, load-balancers can greatly enhance the overall quality of the service offered by a cluster system. Products such as Packeteer [30] offer traffic shaping functionality such that minimum bandwidth guarantees can be allocated to distinct clients or applications. More sophisticated products such as Netscaler [29] apply intelligent connection management that protects the internal cluster nodes from overload in response to large bursts of incoming traffic. However, existing solutions are not aimed at providing throughput and response time guarantees, but are mainly designed to enhance the overall system performance. Futhermore, these techniques rely on the proper configuration of the load-balancers by an expert operator who knows and understands the internal operation of the site to be protected. As such, these are static configurations that are highly tuned for specific settings and that must be repeated for any change occuring in the site's internals. Quorum differs from these approaches in that it guarantees QoS in terms of both throughput and response times. At the same time Quorum does not need to be configured explicitly or tuned by an expert for the specifics of the hardware or software of the site.

At the **operating systems** level, the QoS challenge is typically addressed in terms of resource management. Many research operating systems [6, 10, 37] achieve tight control on the utilization of resources as a way of enforcing capacity isolation between service classes. Although these techniques have proven to be effective in terms of capacity isolation, they are not designed to provide response time guarantees. Furthermore, these techniques control the resources within a single machine and thus cannot be easily extended to clustered environments. One notable exception is Cluster Reserves [4] – a single-node approach that has been scaled to span clustered resources. Although this technique is shown to provide resource isolation at the cluster level, like its single-machine counterparts, it does not provide response time guarantees. Quorum is also a cluster-wide QoS solution that provides both capacity and response time isolation as well as throughput and response time guarantees. It also differs from systems such as Cluster Reserves in that it does not require customization of the operating system used by the cluster's internal nodes.

Middleware systems such Neptune [34, 33] or Application Server [38, 7] include QoS functionality as part of a distributed and potentially scalable infrastructure. By programming the applications to use these primitives it is possible to construct distributed services that offer cluster-wide QoS guarantees. However in order for these frameworks to be effective each of the constituents of a service must be integrated with the middleware infrastructure. This often poses a very restrictive constraint given the heterogeneity and proliferation of current Internet services. Similar approaches that embed the QoS logic directly at the **application level** have also been proposed. For example, the approach presented in SEDA [40] advocates the use of a specific framework for constructing well-conditioned scalable services and [39] shows the effectiveness of this framework when explicit QoS mechanisms are built to prevent overload in busy Internet servers. Rather than building an application with QoS support, other work has modified existing applications to include QoS capabilities [2, 26]. For example, the work done in [2] shows how it is possible to modify the popular Apache web server to provide differentiated services without the use of resource management primitives at the operating system level. However, as is the case with middleware approaches, the large cost of modifying the application code to include QoS mechanisms is only effective if the entirety of the software deployment is able to function in a concerted way towards providing QoS. With Quorum, the applications hosted in an Internet site do not need to be modified or designed for any particular operating system or middleware infrastructure and can directly be used in their native non-QoS state.

Some recent work has investigated resource management techniques using **non-invasive approaches**. Façade [27] is a prototype implementation of a storage controller that throttles I/O requests to a (black-box) disk array. Similar to Quorum, it provides response time isolation (but no throughput isolation) for different I/O streams. However, response time guarantees can only be enforced as long as the total incoming load is below the capacity of the disk array (i.e., no dropping mechanism is implemented). In [25], Jin et al. analyze the effectiveness of several share-based scheduling techniques for differentiating service quality in networked servers. Some of the project goals are similar in nature to Quorum, however the analysis is done only through simulation, focuses only on storage server facilities and does not include a performance study in dynamic scenarios. Furthermore, the devised method is somewhat invasive since it requires offline profiling of the workload and more importantly assumes that the cost of every single request can be known at scheduling time. Other work such as Gatekeeper [18] proposes a proxy system, much like Quorum, that implements admission control for e-commerce applications. However, Gatekeeper is not designed to provide any QoS guarantees, but targeted to reduce the overall response times and improve the performance of the system. Furthermore, it has only been tested in reduced size systems, it targets database backends and relies on extensive profiling of the service applications.

7 Conclusions and Future Work

Commercial Internet service provisioning depends increasingly on the ability to offer differentiated classes of service to groups of potentially competing clients. In addition, the services themselves may impose minimum QoS requirements for correct functionality. However, providing reliable QoS guarantees in large-scale Internet settings is a daunting task. Simple over-provisioning and physical partitioning of resources can be effective but inefficient. Invasive software approaches overcome the inefficiency problem but at the expense of reprogramming and/or re-engineering of the services within a site to implement QoS functionality.

In this paper we present an alternative, non-invasive software approach called Quorum that provides efficient QoS provisioning for Internet services while allowing new levels of flexibility that current service providers require. The presented system functions at the border of an Internet site and uses traffic shaping, admission control, and response feedback to treat the site as a "black-box" control system. Quorum intercepts the request and response streams entering and leaving a site to gauge how and when new requests should be forwarded to the hosted services to ensure throughput and response time guarantees.

We demonstrate the capabilities of our Quorum implementation by experimentally comparing it to the best state-of-the-practice and state-of-the-art approaches. Our results show that, despite being non-invasive, Quorum can enforce the same QoS guarantees as either of the compared techniques, while achieving better resource utilization than over-provisioning and without the application rewriting overhead required by intrusive software approaches. We also demonstrate that our implementation can successfully handle extreme situations such as sudden traffic surges, application misbehavior or node failures. Further, we also demonstrate the powerful flexibility of Quorum by providing QoS guarantees for a complex and heterogeneous Internet service that suffers the same type of harmful conditions. At present, we know of no other published infrastructure that can provide QoS under these challenging conditions. Encouraged by the performance of our results we are currently working on both enhancing the performance and scalability of the Quorum engine as well as improving our algorithms with

more sophisticated control mechanisms. Also we are interested in deploying Quorum on a wider array of Internet services including real commercial sites.

Acknowledgements

This research was supported by the National Science Foundation (grant ITR-0220139), University of California MICRO (grant 02-067) and Sun Microsystems.

References

[1] Akamai Technologies, Inc. Content Delivery Network. http://www.akamai.com.

[2] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing Differentiated Quality-of-Service in Web Hosting Services. In *Proceedings of the First Workshop on Internet Server Performance*, June 1998.

[3] Apache HTTP server. http://httpd.apache.org.

[4] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster Reserves: A Mechanism for Resource Management in Cluster-based Network Servers. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, Santa Clara, California, June 2000.

[5] Ask Jeeves Search Engine. http://www.ask.com.

[6] G. Banga, P. Druschel, and J. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, 1999.

[7] BEA Weblogic Server. http://www.bea.com.

[8] N. Bhatti and R. Friedrich. Web Server Support for Tiered Services. *IEEE Network*, September 1999.

[9] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. RFC 2475, Dec. 1998.

[10] J. Blanquer, J. Bruno, E. Gabber, M. Mcshea, B. Özden, and A. Silberschatz. Resource Management for QoS in Eclipse/BSD. In *Proceedings of the First FreeBSD Conference*, Berkeley, California, Oct. 1999.

[11] J. Blanquer and B. Özden. Fair Queuing for Aggregated Multiple Links. In *Proceedings of the ACM SIGCOMM*, San Diego, CA, August 2001.

[12] J. M. Blanquer, A. Batchelli, K. Schauser, and R. Wolski. Quorum: Providing Flexible Quality of Service for Large-Scale Internet Services. In *UCSB Computer Science, Technical Report TR-2004-27*, 2004.

[13] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: An Overview. RFC 1633, July 1994.

[14] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel. Performance Comparison of Middleware Architectures for Generating Dynamic Web Content. In *ACM/IFIP/USENIX International Middleware Conference*, Rio de Janeiro, Brazil, Jun 2003.

[15] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and Scalability of EJB Applications. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Seattle, Washington, Nov 2002.

[16] CNN.com Article: Internet Proves Vital Communication Tool. http://archives.cnn.com/2001/tech/internet/09/12/attacks.internet/.

[17] A. Demers, S. Keshav, and S. Shenker. Design and Simulation of a Fair Queuing Algorithm. In *Proceedings of the ACM SIGCOMM*, Austin, Texas, September 1989.

[18] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel. A Method for Transparent Admission Control and Request Scheduling in E-Commerce Web Sites. In *Proceedings of the International World Wide Web Conference*, New York, May 2004.

[19] F5 Networks. Big/IP Load Balancer. http://www.f5.com.

[20] Foundry Networks, Inc. Server Iron Internet Traffic Management. http://www.foundrynet.com.

[21] Google. Internet Search Engine. http://www.google.com.

[22] P. Gupta and N. McKeown. Algorithms for Packet Classification. *IEEE Network Special Issue*, March 2001.

[23] P. Hochmuth. Speedy Returns are Google's goal. *Network World Fusion article*, Jan 2003. http://www.nwfusion.com/news/2003/0901google.html.

[24] Java NIO. http://java.sun.com/j2se/1.5.0/docs/guide/nio.

[25] W. Jin, J. Chase, and J. Kaur. Interposed Proportional Sharing for a Storage Service Utility. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, New York, June 2004.

[26] Y. Lu, T. F. Abdelzaher, and A. Saxena. Design, Implementation, and Evaluation of Differentiated Caching Services. *IEEE Transactions on Parallel and Distributed Systems*, May 2004.

[27] C. Lumb, A. Merchant, and G. Alvarez. Façade: Virtual Storage Devices with Performance Guarantees. In *Proceedings of the USENIX Conference on File and Storage Technologies*, San Francisco, CA, March 2003.

[28] D. Mitra, R. Gibbens, and B. D. Huang. State-Dependent Routing on Symmetric Loss Networks with Trunk Reservations I. *IEEE/ACM Transactions on Communications*, February 1993.

[29] Netscaler, Inc. Request Switching Technology. http://www.netscaler.com.

[30] Packeteer, Inc. Application Traffic Management System. http://www.packeteer.com/.

[31] A. K. Parekh and R. G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks-the Single Node Case. *IEEE/ACM Transactions on Networking*, June 1993.

[32] RUBiS Rice University Bidding System. http://rubis.objectweb.org.

[33] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated Resource Management for Cluster-based Internet Services. In *Proc. of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[34] K. Shen, T. Yang, L. Chu, J. Holliday, D. Kuschner, and H. Zhu. Neptune: Scalable Replication Management and Programming Support for Cluster-based Network Services. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, San Francisco, CA, Mar 2001.

[35] Teoma. Internet Search Engine. http://www.teoma.com.

[36] Apache Tomcat Server. http://jakarta.apache.org/tomcat.

[37] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel Mechanisms for Service Differentiation in Overloaded Web Servers. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, Massachusetts, June 2001.

[38] WebSphere Extended Deployment. http://www.ibm.com/software/websphere/.

[39] M. Welsh and D. Culler. Overload Management as a Fundamental Service Design Primitive, Saint-Emilion, France Sept 2002.

[40] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, Banff, Canada, Oct 2001.

[41] H. Zhu, H. Tang, and T. Yang. Demand-driven Service Differentiation in Cluster-based Network Servers. In *Proceedings of the IEEE INFOCOM*, Anchorage, Alaska, April 2001.

Trickles: A Stateless Network Stack for Improved Scalability, Resilience, and Flexibility

Alan Shieh Andrew C. Myers Emin Gün Sirer

Dept. of Computer Science
Cornell University
Ithaca, NY 14853
{ashieh,andru,egs}@cs.cornell.edu

Abstract

Traditional operating system interfaces and network protocol implementations force system state to be kept on both sides of a connection. Such state ties the connection to an endpoint, impedes transparent failover, permits denial-of-service attacks, and limits scalability. This paper introduces a novel TCP-like transport protocol and a new interface to replace sockets that together enable all state to be kept on one endpoint, allowing the other endpoint, typically the server, to operate without any per-connection state. Called Trickles, this approach enables servers to scale well with increasing numbers of clients, consume fewer resources, and better resist denial-of-service attacks. Measurements on a full implementation in Linux indicate that Trickles achieves performance comparable to TCP/IP, interacts well with other flows, and scales well. Trickles also enables qualitatively different kinds of networked services. Services can be geographically replicated and contacted through an anycast primitive for improved availability and performance. Widely-deployed practices that currently have client-observable side effects, such as periodic server reboots, connection redirection, and failover, can be made transparent, and perform well, under Trickles. The protocol is secure against tampering and replay attacks, and the client interface is backwards-compatible, requiring no changes to sockets-based client applications.

1 Introduction

The flexibility, performance, and security of networked systems depend in large part on the placement and management of system state, including both the kernel-level and application-level state used to provide a service. A critical issue in the design of networked systems is where to locate, how to encode, and when to update system state. These three aspects of network protocol stack design have far reaching ramifications: they determine protocol functionality, dictate the structure of applications, and may enhance or limit performance.

Consider a point-to-point connection between a web client and server. The system state consists of TCP protocol parameters, such as window size, RTT estimate, and slow-start threshold, as well as application-level data, such as user id, session id, and authentication status. There are only three locations where state can be stored, namely, the two endpoints and the network in the middle. While the end-to-end argument provides guidance on where not to place state and implement functionality, it still leaves a considerable amount of design flexibility that has remained largely unexplored.

Traditional systems based on sockets and TCP/IP distribute session state across *both* sides of a point-to-point connection. Distributed state leads to three problems. First, connection failover and recovery is difficult, non-transparent, or both, as reconstructing lost state is often non-trivial. Web server failures, for instance, can lead to user-visible connection resets. Second, dedicating resources to keeping state invites denial of service (DoS) attacks that use up these resources. Defenses against such attacks often disable useful functionality: few stacks accept piggybacked data on SYN packets, increasing overhead for short connections, and Internet servers often do not allow long-running persistent HTTP connections, increasing overhead for bursty accesses [8]. Finally, state in protocol stacks limits scalability: servers cannot scale up to large numbers of clients because they need to commit per-client resources, and similarly cannot scale down to tiny embedded devices, as there is a lower bound on the resources needed per connection.

In this paper, we investigate a fundamentally different way to structure a network protocol stack, in which system state can be kept entirely on one side of a network connection. Our Trickles protocol stack enables encapsulated state to be pushed from the server to the client. The client then presents this state to the server when requesting service in subsequent packets to reconstitute the server-side state. The encapsulated state thus acts as a form of *network continuation* (Figure 1). A new

server-side interface to the network protocol stack, designed to replace sockets, allows network continuations to carry both kernel and application level state, and thus enables stateless network services. On the client side, a compatibility layer ensures that sockets-based clients can transparently migrate to Trickles. The use of the TCP packet format at the wire level reduces disruption to existing network infrastructure, such as NATs and traffic shapers, and enables incremental deployment.

A stateless network protocol interface and implementation have many ramifications for service construction. Self-describing packets carrying encapsulated server state enable services to be replicated and migrated between servers. Failure recovery can be instantaneous and transparent, since redirecting a continuation-carrying Trickles packet to a live server replica will enable that server to respond to the request immediately. In the wide area, Trickles obviates the key concern about the suitability of anycast primitives [3] for stateful connection-oriented sessions by eliminating the need for route stability. Server replicas can thus be placed in geographically diverse locations, and satisfy client requests regardless of their past communications history. Eliminating the client-server binding obviates the need for DNS redirection and reduces the potential security vulnerabilities posed by redirectors. In wireless networks, Trickles enables connection suspension and migration to be performed without recourse to intermediate nodes in the network to temporarily hold state.

A stateless protocol stack can rule out many types of denial-of-service attacks on memory resources. While previous work has examined how to thwart DoS attacks targeted at specific parts of the transport protocol, such as SYN floods, Trickles provides a general approach applicable for all attacks against kernel and application-level state.

Overall, this paper makes three contributions. First, it describes the design and implementation of a network protocol stack that enables all per-connection state to be safely migrated to one end of a network connection. Second, it outlines a new TCP-like transport protocol and a new application interface that facilitates the construction of event-driven, continuation-based applications and fully stateless servers. Finally, it demonstrates through a full implementation that applications based on this infrastructure achieve performance comparable to that of TCP, interact well with other TCP-friendly flows, and scale well.

The rest of the paper describes Trickles in more detail. Section 2 describes the Trickles transport protocol. Section 3 presents the new stateless server API, while Section 4 describes the behavior of the client. Section 5 presents optimizations that can significantly increase the performance of Trickles. Section 6 evaluates our Linux implementation. Section 7 discusses related work and Section 8 summarizes our contributions and their implications for server design.

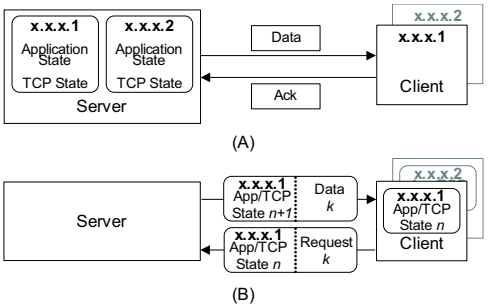


Figure 1: TCP versus Trickles state. (A) TCP holds state at server, even for idle connection x.x.x.2. (B) Trickles encapsulates and ships server state to the client.

2 Stateless Transport Protocol

The Trickles transport protocol provides a reliable, high-performance, TCP-friendly stream abstraction while placing per-connection state on only one side of the connection. Statelessness makes sense when connection characteristics are asymmetric; in particular, when a high-degree node in the graph of sessions (typically, a server) is connected to a large number of low-degree nodes (for example, clients). A stateless high-degree node would not have to store information about its many neighbors. For this reason we will refer to the stateless side of the connection as the *server* and the stateful side as the *client*, though this is not the only way to organize such a system.

To make congestion-control decisions, the stateless side needs information about the state of the connection, such as the current window size and prior packet loss. Because the server does not keep state about the connection, the client tracks state on the server's behalf and attaches it to requests sent to the server. The updated connection state is attached to response packets and passed to the client. This piggybacked state is called a *continuation* because it provides the necessary information for the server to later resume the processing of a data stream.

The Trickles protocol simulates the behavior of the TCP congestion control algorithm by shipping the kernel-level state, namely the TCP control block (*TCB*), to the client side in a *transport continuation*. The client ships the transport continuation back to the server in each packet, enabling the server protocol stack to regenerate state required by TCP congestion control [26]. Trickles also supports stateless user-level server applications; to permit a server application to suspend processing without retaining state, the application may add an analogous

user continuation to outgoing packets.

During the normal operation of the Trickles protocol, the client maintains a set of user and transport continuations. When the client is ready to transmit or request data, it generates a packet containing a transport continuation, any loss information not yet known to the server, a user continuation, and any user-specified data. On processing the request, the server protocol stack uses the transport continuation and loss information to compute a new transport continuation. The user data and user continuation are passed to the server application, along with the allowed response size. The user continuation and data are used by the application to compute the response.

With Trickles, responsibility for data retransmission lies with the server application, since a reliable queuing mechanism, such as that found in TCP implementations, is stateful and holds data in a send buffer until it is acknowledged. Therefore, a Trickles server application must be able to reconstruct old data, either by supporting (stateless) reconstruction of previously transmitted data, or by providing its own (stateful) buffering. This design allows applications to control the amount of state devoted to each connection, and share buffer space where possible.

2.1 Transport and user continuations

The Trickles transport continuation encodes the part of the TCB needed to simulate the congestion control mechanisms of the TCP state machine. For example, the continuation includes the packet number, the round trip time (*RTT*), and the slow-start threshold (*ssthresh*). In addition, the client attaches a compact representation of the losses it has incurred. This information enables the server to recreate an appropriate TCB. Transport continuations are $75+12m$ bytes, where m is the number of loss events being reported to the server (usually $m = 1$). Our implementation uses delayed acknowledgments, matching common practice for TCP [1].

The user continuation enables a stateless server application to resume processing in an application-dependent manner. Typically, the application will need information about what data object is being delivered to the client, along with the current position in the data stream. For a web server, this might include the URL of the requested page (or a lower-level representation such as an inode number) and a file offset. Of course, nothing prevents the server application from maintaining state where necessary.

2.2 Security

Migrating state to the client exposes the server to new attacks. It is important to prevent a malicious user or third party from tampering with server state in order to extract an unfair share of the service, to waste bandwidth, to launch a DDoS attack, or to force the server to execute an invalid state [2]. Such attacks might employ two mechanisms: modifying the server state—because it is no longer secured on the server, and performing replay attacks—because statelessness inherently admits replay of old packets.

Maintaining state integrity

Trickles protects transport continuations against tampering with a message authentication code (MAC), signed with a secret key known only to the server and its replicas. The MAC allows only the server to modify protected state, such as *RTT*, *ssthresh*, and window size. Similarly, a server application should protect its state by using a MAC over the user continuation. Malicious changes to the transport or user continuation are detected by the server kernel or application, respectively.

Hiding losses [19, 10] is a well-known attack on TCP that can be used to gain better service or trigger a DDoS attack. Trickles avoids these attacks by attaching unique nonces to each packet. Because clients cannot predict nonce values, if a packet is lost, clients cannot substitute the nonce value for that packet.

Trickles clients signal losses using selective acknowledgment (SACK) proofs, computed from the packet nonces, that securely describe the set of packets received. The nonces are grouped by contiguous ranges, and are compressed into a compact range summary that can be checked efficiently. Let p_i be packet i's nonce. The range $[m, n]$ is summarized by XORing together each p_i in the range into a single word. Imposing additional structure on the nonces enables Trickles to generate per-packet nonces and to verify multi-packet ranges in $O(1)$ time. Define a sequence of random numbers $r_x = f(K, x)$, where f is a keyed cryptographic hash function. If $p_i = r_i \oplus r_{i+1}$, then $p_1 \oplus p_2 \oplus ... \oplus p_n = r_1 \oplus r_{n+1}$. Thus, the server can generate and verify nonces with a constant number of r_x computations. Trickles distinguishes retransmitted packets from the original by using a different server key K' to derive retransmitted nonces. This suffices to keep an attacker from using the nonce from the retransmitted packet to forge a SACK proof that masks the loss of the original packet.

Note that this nonce mechanism protects against omission of losses but not against insertion of losses; as in TCP, a client that pretends not to receive data is self-limiting because its window size shrinks.

Protection against replay

Stateless servers are inherently vulnerable to replay attacks. Since the behavior of a stateless system is independent of history, two identical packets will elicit the same response. Therefore, protection against replay requires some state. For scalability, this extra state should

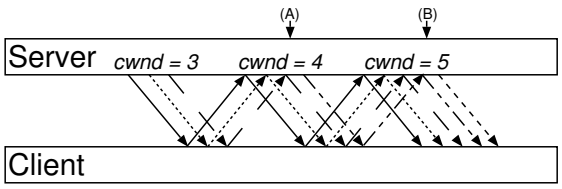


Figure 2: A sample TCP or Trickles connection. Each line pattern corresponds to a different trickle. Initially, there are $cwnd$ trickles. At points where $cwnd$ increases (A, B), trickles are split.

be small and independent of the number of connections. One possible replay defense is a simple hash table keyed on the transport continuation MAC. This bounds the effect of a replay attack, and if hash collisions indicate the presence of an attack, the size of the hash table can be increased. The hash table records recent packet history up until a time horizon. Each transport continuation carries a server-supplied timestamp that is checked on packet arrival; packets older than the time horizon are simply discarded. Since the timestamp generation and freshness check are both performed on the server, clock synchronization between the client and server is not necessary. The growth of the hash table is capped by periodically purging and rebuilding it to capture only the packets within the time horizon.

The replay defense mechanism can be implemented in the server kernel or in the server application. The advantage of detecting replay in the kernel is that duplicate packets can be flagged early in processing, reducing the strain on the kernel-to-application signaling mechanism. Placing the mechanism in the application is more flexible, because application-specific knowledge can be applied. For simplicity and flexibility, we have chosen to place replay defense in the application. In either case, Trickles is more robust against state-consumption attacks than TCP.

2.3 The trickle abstraction

Figure 2 depicts the exchange of packets between the two ends of a typical TCP or Trickles connection. For simplicity, there is no loss, packets arrive in order, and delayed acknowledgments are not used. Except where the current window size ($cwnd$) increases (at times A and B), the receipt of one packet from the client enables the server to send one packet in response, which in turn triggers another packet from the client, and so on. This sequence of related packets forms a *trickle*.

A trickle captures the essential control and data flow properties of a stateless server. If the server does not remember state between packets, information can only flow forward along individual trickles and so the response of the server to a packet is solely determined by the incoming trickle. A stream of packets is decomposed into multiple disjoint trickles; each packet is a member of exactly one trickle. These trickles can be thought of as independent threads that exchange information only on the client side.

In the Trickles protocol, the congestion control algorithm at the server operates on each trickle independently. These independent instances cooperate to mimic the congestion control behavior of TCP. At a given time there are $cwnd$ simultaneous trickles. When a packet arrives at the server, there are three possible outcomes. In the common case, Trickles permits the server application to send one packet in response, *continuing* the current trickle. However, if packets were lost, the server may *terminate* the current trickle by not permitting a response packet; trickle termination reduces the current window size ($cwnd$) by 1. The server may also increase $cwnd$ by *splitting* the current trickle into $k > 1$ response packets, and hence begin $k - 1$ new trickles.

Split and *terminate* change the number of trickles and hence the number of possible in-flight packets. Congestion control at the server consists of using the client-supplied SACK proof to decide whether to continue, terminate, or split the current trickle. Making Trickles match TCP's window size therefore reduces to splitting or terminating trickles whenever the TCP window size changes. When processing a given packet, Trickles simulates the behavior of TCP at the corresponding acknowledgment number based on the SACK proof, and then split or terminate trickles to generate the same number of response packets. The subsequent sections describe how to statelessly perform these decisions to match the congestion avoidance, slow start, and fast retransmit behavior of TCP.

2.4 Trickle dataflow constraints

Statelessness complicates matching TCP behavior, because it fundamentally restricts the data flow allowed between the processing of different packets. This restriction is the main source of complexity in designing a stateless transport protocol.

Because Trickles servers are stateless, the server forgets all the information for a trickle after processing the given packet, whereas TCP servers retain this state persistently in the TCB. Consider the comparison in Figure 3, illustrating what happens when two packets from the same connection are received in succession. For Trickles, the state update from processing the first packet is not available when the second packet is processed at the point (B). At the earliest, this state update can be made available at point (D) in the figure, after a round trip through the client, during which the client fuses packet loss information from the two server responses and sends that information back with the second trickle. This ex-

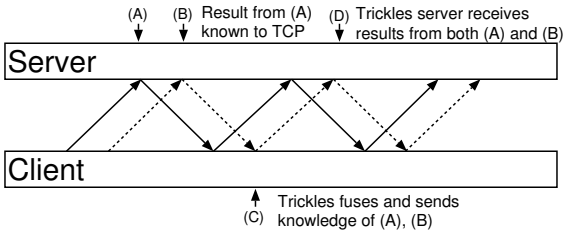


Figure 3: Difference in state/result availability between TCP and Trickles. TCP server knows the result of processing (A) earlier than Trickles server.

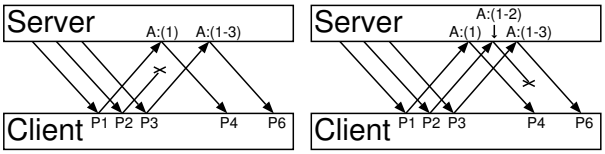


Figure 4: Equivalence of reverse and forward path loss in Trickles. Due to dataflow constraints, the packet following a lost packet does not compensate for the loss immediately. Neither the server nor the client can distinguish between (A) and (B). The loss will be discovered through subsequent SACK proofs.

ample illustrates that server state cannot propagate directly between the processing of consecutive packets, but is available to server-side processing a round trip later.

The round-trip delay in state updates makes it challenging to match the congestion control action of TCP. Trickles circumvents the delay by using prediction. When a packet arrives at the server, the server can only know about packet losses that happened one full window earlier. It optimistically assumes that all packets since that point have arrived successfully, and accordingly makes the decision to continue, split, or terminate. Optimism makes the common case of infrequent packet loss work well.

Concurrent trickles must respond consistently and quickly to loss events. By providing each trickle with the information needed to predict the actions of other trickles, redundant operations are avoided. Since the client-provided SACK proofs control trickle behavior, we impose an invariant on SACK proofs to allow a later trickle to infer the SACK proof of a previous trickle: given a SACK proof L, any proof L' sent subsequently contains L as a prefix. This *prefix property* allows the server to predict SACK proofs prior to L by simply computing a prefix. Conceptually, SACK proofs cover the complete loss history, starting from the beginning of the connection. As an optimization to limit the proof size, a Trickles server allows the client to omit initial portions of the SACK proof once the TCB state fully reflects the server's response to those losses. This is guaranteed to occur after all loss events, once recovery or retransmission timeout finishes.

With any prediction scheme, it is sometimes necessary to recover from misprediction. Suppose a packet is lost before it reaches the server. Then the server does not generate the corresponding response packet. This situation is indistinguishable from a loss of the response on the server to client path: in both cases, the client receives no response (Figure 4). Consequently, a recovery mechanism for response losses also suffices to recover from request packet losses, simplifying the protocol. Note, however, that Trickles is more sensitive to loss than TCP. While TCP can elide some ACK losses with implicit acknowledgments, such losses in Trickles require retransmission of the corresponding request and data.

2.5 Congestion control algorithm

We are now equipped to define the per-trickle congestion control algorithm. The algorithm operates in three modes that correspond to the congestion control mechanisms in TCP Reno [1]: congestion avoidance/slow start, fast recovery, and retransmit timeout. Trickles strives to emulate the congestion control behavior of TCP Reno as closely as possible by computing the target $cwnd$ of TCP Reno, and performing split or terminate operations as needed to move the number of trickles toward this target. Between modes, the set of valid trickles changes to reflect the increase or decrease in $cwnd$. In general, the number of trickles will decrease in a mode transition; the valid trickles in the new mode are known as *survivors*.

Slow start and congestion avoidance

In TCP Reno, slow start increases $cwnd$ by one per packet acknowledgment, and congestion avoidance increases $cwnd$ by one for every window of acknowledgments. Trickles must determine when TCP would have increased $cwnd$ so that it can properly split the corresponding trickle. To do so, Trickles associates each request packet with a request number k, and uses the function $\texttt{TCPCwnd}(k)$ to map from request number k to TCP $cwnd$, specified as a number of packets. Abstractly, $\texttt{TCPCwnd}(k)$ executes a TCP state machine using acknowledgments 1 through k and returns the resulting $cwnd$. Given the assumption that no packets are lost, and no ACK reordering occurs, the request number of a packet fully determines the congestion response of a TCP Reno server.

Upon receiving request packet k, the server performs the following trickle update:

- $\texttt{CwndDelta} := \texttt{TCPCwnd}(k) - \texttt{TCPCwnd}(k-1)$
- Generate $\texttt{CwndDelta} + 1$ responses: continue the original trickle, and split $\texttt{CwndDelta}$ times.

$$\texttt{TCPCwnd}(k) = \begin{cases} startCwnd + (k - TCPBase) & \textbf{if } k < A \\ ssthresh & \textbf{if } A \le k < A + ssthresh \\ F(k - A) + 1 + ssthresh & \textbf{if } A + ssthresh \le k \end{cases}$$

where

$$A = ssthresh - startCwnd + TCPBase$$

and $F(N)$ is the largest integer less than the positive value of x that is a zero of

$$\frac{x(x+1) - ssthresh(ssthresh+1)}{2} - N$$

Figure 5: Closed-form solution of TCP simulation.

Assuming $\texttt{TCPCwnd}(k)$ is a monotonically increasing function, which is indeed the case with TCP Reno, this algorithm maintains $cwnd$ trickles per RTT, precisely matching TCP's behavior. If $\texttt{TCPCwnd}(k)$ were implemented with direct simulation as described above, it would require $O(n)$ time per packet, where n is the number of packets since connection establishment. Fortunately, for TCP Reno, a straightforward strength reduction yields the closed-form solution shown in Figure 5, which can be computed in $O(1)$ time.

The $\texttt{TCPCwnd}(k)$ formula is directly valid only for connections where no losses occur. A connection with losses can be partitioned at the loss positions into multiple pieces without losses; $\texttt{TCPCwnd}(k)$ is valid within each individual piece. The free parameters in $\texttt{TCPCwnd}(k)$ are used to adapt the formula for each piece: $startCwnd$ and $ssthresh$ are initial conditions at the point of the loss, and $TCPBase$ corresponds to the last loss location.

Fast retransmit/recovery

In fast retransmit/recovery, TCP Reno uses duplicate acknowledgments to infer the position of a lost packet (Figure 6). The lost packet is retransmitted, the $cwnd$ is halved, and transmission of new data temporarily squelched to decrease the number of in-flight packets to $newCwnd$. Likewise, Trickles uses its SACK proof to infer the location of lost packets, retransmits these packets, halves the $cwnd$, and terminates a sufficient number of trickles to deflate the number of in-flight packets to $newCwnd$ (Figure 7).

Fast retransmit/recovery is entered when the SACK proof contains a loss. A successful fast retransmit/recovery phase is followed by a congestion avoidance phase. Since multiple trickles must execute the algorithm in parallel, each with a different recovery role, the SACK prefix property is critical to proper operation, as it allows each trickle to predict the input and recovery action of preceding trickles. A client that violates the prefix property in packets it sends to the server will cause inconsistent computations on the server side, and may receive data and transport continuations redundantly or not receive them at all.

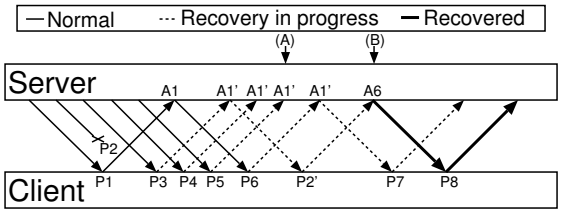


Figure 6: TCP recovery. Duplicate ACKs signal recovery (A). Subsequent ACKs are ignored until number of outstanding packets drops to new $cwnd$. Recovery ends when client acknowledges all packets (B).

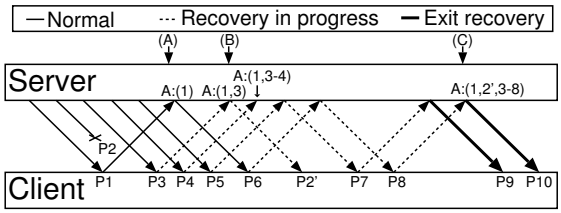


Figure 7: Trickles Recovery. First packet following loss triggers a retransmission (A). Trickles are subsequently terminated to deflate $cwnd$ (B). Recovery ends when $cwnd$ survivors are generated; $cwnd$ has dropped from the original value of 5 to 2 (C).

For request packet with packet number k during fast retransmit/recovery mode, Trickles performs the following operations:

1. ```
   firstLoss := sequence number of
                first loss in input
   cwndAtLoss := TCPCwnd(firstLoss - 1)
   lossOffset := k - firstLoss
   newCwnd := numInFlight / 2
   ```

   The protocol variable `firstLoss` is derived from the SACK proof. The SACK proofs for the trickle immediately after a loss, as well as all subsequent trickles before recovery, will report a gap. The SACK prefix invariant ensures that each trickle will compute consistent values for the protocol variables shown above.

2. If `k` acknowledges the first packet after a run of losses, retransmit the lost packets (Figure 7). This is required to achieve the reliable delivery guarantees of TCP. A `burstLimit` parameter, similar to that suggested for TCP [1], limits the number of packets that may be retransmitted in this manner; losses beyond `burstLimit` are handled via a timeout and not via fast retransmit.
   ```

3. The goal in fast retransmit is to terminate `n = cwndAtLoss - newCwnd` trickles, and generate `newCwnd` survivor trickles. We choose to terminate the first `n` trickles, and retain the last `newCwnd` trickles using the following algorithm:

   (a) If `cwndAtLoss - lossOffset < newCwnd`, continue the trickle. Otherwise, terminate the trickle. (b) If `k` immediately follows a run of losses, generate the trickles for all missing requests that would have satisfied (a).

   Test (a) deflates the number of trickles to `newCwnd`. First, a sufficient number of trickles are terminated to drop the number of trickles to `newCwnd`. Then, all subsequent trickles become *survivors* that will bootstrap the subsequent slow start/congestion avoidance phase. If losses occur while sending the surviving trickles to the client, then the number of outstanding trickles will fall below `newCwnd`. So condition (a) guarantees that the new window size will not exceed the new target, while condition (b) ensures that the new window will meet the target.

   Note that when the server decides to recreate multiple lost trickles per condition (b), it will not have access to corresponding user continuations for the lost packets. Consequently, the server transport layer cannot invoke the application and generate the corresponding data payload. Instead, the server transport layer simply generates the transport continuations associated with the lost trickles and ships them to the client as a group. The client then regenerates the trickles by retransmitting these requests to the server with matching user continuations.

Following fast recovery, the simulation initial conditions are updated to reflect the conditions at the recovery sequence number: `TCPBase` points to the recovery point, and `ssthresh = startCwnd = newCwnd`, reflecting the new window size.

Retransmit timeout

During a retransmit timeout, the TCP Reno sender sets $ssthresh = cwnd/2$, $cwnd = InitialCwnd$, and enters slow start. In Trickles, the client kernel is responsible for generating the timeout, as the server is stateless and cannot keep such a timer. Let *firstLoss* be the first loss seen by the client since the last retransmit timeout or successful recovery. For a retransmission timeout request, the server executes the following steps to initiate slow start:

1. ```
   a := firstLoss
   ssthresh := TCPCwnd(a-1)/2
   cwnd := InitialCwnd
   ```
2. Split $cwnd - 1$ times to generate $cwnd$ survivors. Set `TCPCwnd`(k) initial conditions to equivalent TCP post-recovery state.

2.6 Compatibility with TCP

Trickles is backward compatible with TCP in several important ways, making it possible to incrementally adopt Trickles into the existing Internet infrastructure. Compatibility at the network level, due to similar wire format, similar congestion control algorithm, and TCP-friendly behavior, ensures interoperability with routers, traffic shapers, and NAT boxes.

The client side of Trickles provides to the client application a standard Berkeley sockets interface, so the client application need not be aware of the existence of Trickles, and only the client kernel needs modification

Trickles-enabled clients are compatible with existing TCP servers. The initial SYN packet from a Trickles client carries a TCP option to signal the ability to support Trickles. Servers that are able to support Trickles respond to the client with a Trickles response packet, and a Trickles connection proceeds. Servers that understand only TCP respond with a standard TCP SYN-ACK, causing the client to enter standard TCP mode.

A Trickles server can also be compatible with standard TCP clients, by handling standard TCP requests according to the TCP protocol. Of course, the server cannot be stateless for those clients, so some servers may elect to support only Trickles.

3 Trickles server API

The network transport protocol described in Section 2 makes it possible to maintain a reliable communications channel between a client and server with no per-connection state in the server kernel. However, the real benefit of statelessness is obtained when the entire server is stateless. The Trickles server API allows servers to offload user-level state to the client, so that the server machine maintains no state at any layer of the network stack.

3.1 The event queue

In the Trickles server API, the server application does not communicate using per-connection file descriptors, as these would entail per-connection state. Instead, the API exports a queue of transport-level *events* to the application. For example, client data packets and ACKs appear as events. Since Trickles is stateless, events only occur in response to client packets.

Upon processing a client request packet, the Trickles transport layer may either terminate the trickle, or continue the associated trickle and split off zero or more trickles. If the transport generates a response, a single event is passed to the application, describing the incoming packet and all the response trickles. The event includes all the data from the request and also the user continuation from the request to the application. API state is linear in the number of unprocessed requests, which
   ```

```
msk_send(int fd, minisock *, char *, size_t);
msk_sendv(int fd, minisock *, tiovec *, int);
msk_sendfilev(int fd, minisock *, fiovec *, int);
msk_setucont(int fd, minisock *, int pkt,
                  char* buf, size_t);
msk_sendbulk(int fd, mskdesc *, int len);
msk_drop(int fd, minisock *);
msk_detach(int fd, minisock *);
msk_extract_events(int fd, extract_mskdesc_in *,
           int inlen, msk_collection *, int *outlen);
msk_install_events(int fd, msk_collection *, int);
msk_request(int fd, char *req, int reqlen,
           int reservelen);
```

Figure 8: The minisocket API.

is bounded by the ingress bandwidth. The event queue eliminates a layer of multiplexing and demultiplexing found in the traditional sockets API that can cause excess processing overhead [4].

To avoid copying of events, the event queue is a synchronization-free linked list mapped in both the application and kernel; it is mapped read-only in the application, and can be walked by the application without holding locks. While processing requests, the kernel allocates all per-request data structures in the shared region.

3.2 Minisockets

The Trickles API object that represents a remote endpoint is called a *minisocket*. Minisockets are transient descriptors that are created when an event is received, and destroyed after being processed. Like standard sockets, each minisocket is associated with one client, and can send and receive data. Operationally, a minisocket acts as a transient TCP control block, created from the transport continuation in the associated packet. Because the minisocket is associated with a specific event, the extent of each operation is more limited. Receive operations on the minisocket can only return input data from the associated event, and send operations may not send more data than is allowed by congestion control. Trickles delivers *OPEN*, *REQUEST*, and *CLOSE* events when connections are created, client packets are received, and clients disconnect, respectively.

3.3 Minisocket operations

The minisocket API is shown in Figure 8. Minisockets are represented by the structure `minisock *`. All minisockets share the same file descriptor (`fd`), that of their listen (server) socket. To send data with a minisocket, applications use `msk_send`. It copies packet data to the kernel, constructs and sends Trickles response packets, then deallocates the minisocket. `msk_setucont` allows the application to install user continuations on a per-packet basis. Trickles also provides scatter-gather, zero copy, and packet batch processing interfaces.

Allowing servers to directly manipulate the minisocket queue enables new functionality not possible with sockets. Requests sent to a node in a cluster can be redirected to a different node holding a cached copy, without breaking the connection. During a denial of service attack, a server may elect to ignore events altogether. The event management interface enables such manipulations of the event queue. While these capabilities are similar to those proposed in [17] for TCP, Trickles can redistribute events at a packet-level granularity.

The `msk_extractEvents` and `msk_insertEvents` operations manipulate the event queue to extract or insert minisockets, respectively. The extracted minisockets are protected against tampering by MACs. Extracted minisockets can be migrated safely to other sockets, including those on other machines.

4 Client-side processing

A Trickles client stack implements a Berkeley sockets interface using the Trickles transport protocol. Thus, the client application need not be aware of the presence of Trickles. The structure of Trickles allows client kernels to use a straightforward algorithm to maintain the transport protocol. The client kernel generates requests using the transport continuations received from the server, while ensuring that the prefix property holds on the sequence of SACK proofs reported to the server. Should the protocol stall, the client times out and requests a retransmission and slow start.

In addition to maintaining the transport protocol, a client kernel manages user continuations, storing new continuations and attaching them to requests as appropriate. For instance, the client must provide all continuations needed to generate a particular data request.

4.1 Standardized user continuations

To facilitate client-side management of continuations, and to simplify server programming, Trickles defines standard user continuation formats understood by servers and clients. These formats encode the mapping between continuations and data requests, and provide a standard mechanism for bootstrapping and generating new continuations.

Two kinds of continuations can be communicated between the client and server: *output* continuations that the server application uses to resume generating output to the client at the correct point in the server's output stream, and *input* continuations that the server application uses to help it resume correctly accepting client input. Having separate continuations allows the server to decouple input and output processing.

4.2 Input continuations

When a client sends data to the server, it accompanies the data with an appropriate input continuation, except for the very first packet when no input continuation is needed. For single-packet client requests, an input continuation is not needed. For requests that span multiple packets, an input continuation contains a digest of the data seen thus far. Of course, if the server needs lengthy input from the client yet cannot encode it compactly into an input continuation, the server application will not be able to remain stateless.

If, after receiving a packet from the client, the server application is unable to generate response packets, it sends an updated input continuation back to the client kernel, which will respond with more client data accompanied by the input continuation. The server need not consume all of the client data; the returned input continuation indicates how much input was consumed, allowing the client's transmit queue to be advanced correspondingly. The capability to not read all client data is important because the server may not be able to compactly encode arbitrarily truncated client packets in an input continuation.

4.3 Output continuations

When the server has received sufficient client data to begin processing a request, it provides the client with an output continuation for the response. The client can then use the output continuation to request the response data. For a web server, the output continuation might contain an identifier for the data object being delivered, along with an offset into that data object.

In general, the client kernel will have a number of output continuations available that have arrived in various packets from the server. Client requests include the requested ranges of data, along with the corresponding output continuations. To allow the client to select the correct output continuation, an output continuation includes, in addition to opaque application-defined data, two standard fields, `validStart` and `validEnd`, indicating the range of bytes for which the output continuation can be used to generate data.

The client cannot request an arbitrarily-sized range because the congestion control algorithm restricts the amount of data that may be returned for each request. To compute the proper byte range size, the client simulates the server's congestion control action for a given transport continuation and SACK proof.

5 Optimizations

The preceding sections described the operation of the basic Trickles protocol. The performance of the basic protocol is improved significantly by three optimizations.

5.1 Socket caching

While the basic Trickles protocol is designed to be entirely stateless, and thereby consume little memory, it can be easily extended to take advantage of server memory when available. In particular, the server host need not discard minisockets and reconstitute the server-side TCB from scratch based on the client continuation. Instead, it can keep minisockets for frequently used connections in a server-side cache, and match incoming packets to this pool via a hash table. A cache hit will obviate the need to reconstruct the server-side state or to validate the MAC hash on the client-supplied continuation. When pressed for memory, entries in the minisocket cache can simply be dropped, as minisockets can be recreated at any time. Fundamentally, the cache acts as soft-state that can enable the server to operate in a stateful manner whenever resources permit and reduce the processing burden, while the underlying stateless protocol provides a safety net in case the state needs to be reconstructed from scratch.

5.2 Parallel requests and sparse sequence numbers

The concurrent nature of Trickles enables a second optimization for parallel downloads. Standard TCP operates serially, transmitting streams mostly in-order, and immediately filling any gaps stemming from losses. However, many applications, including web browsers, need to download multiple files concurrently. With standard TCP, such concurrent transactions either require multiple connections, leading to well-documented inefficiencies [15], or complex application-level protocols, such as HTTP 1.1 [11], for framing. In contrast, trickles are inherently concurrent. Concurrency can improve the performance of both fetching and sending data to the server.

The Trickles protocol allows a client application to concurrently request different, non-adjoining sequence number ranges from the server on a single connection. The response packets from the server, which will carry data belonging to different objects distributed through the sequence number space, will nevertheless be subject to a single TCP-friendly flow equation, acting in effect like a single, HTTP/1.1-like flow with application level framing. Since, in some cases, the sizes of the objects may not be known in advance, Trickles clients can conservatively dedicate large regions of the sequence number space to each object. A server response packet may include a SKIP notification that indicates that the object ended before the end of its assigned range. A client receiving a SKIP logically elides the remainder of the object region, without reserving physical buffer space, passing it to applications, or waiting for additional packets from the server. Consequently, the inherent parallelism between Trickles can be used to multiplex logically separate transmissions on a given connection, while subjecting them to the same flow equation.

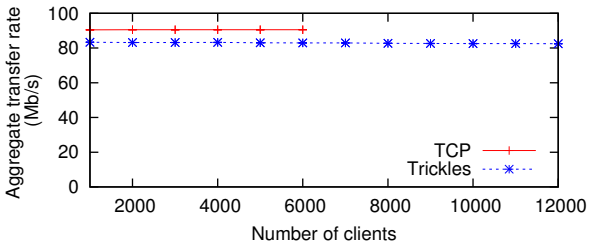


Figure 9: Aggregate throughput for Trickles and TCP. TCP fails for tests with more than 6000 clients.

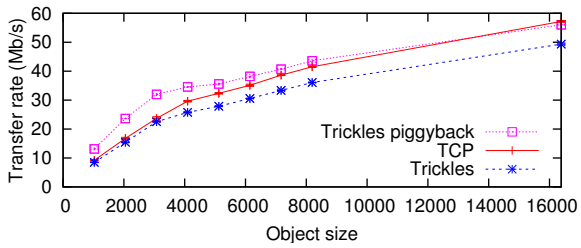


Figure 10: Trickles and TCP throughput for a single, isolated client at various object sizes.

Trickles clients can also send multiple streams of data to the server using the same connection. A stateless server is oblivious to the number of different input sequences on a connection. By performing multiple server input operations in parallel, a client can reduce the total latency of a sequence of such operations.

5.3 Delta encoding

While continuations add extra space overhead to each packet, predictive header compression can be used to drastically reduce the size of the continuations transmitted by the server. Since the Trickles client implementation simulates the congestion control algorithm used by the server, it can predict the server's response. Consequently, the server need only transmit those fields in the transport continuation that the client mispredicts (e.g. a change due to an unanticipated loss), or cannot generate (e.g. timestamps). Of course, the server MAC still needs to be computed and transmitted on every continuation, as the client cannot compute the secure server hash.

6 Evaluation

In this section, we evaluate the quantitative performance of Trickles through microbenchmarks, and show that it performs well compared to TCP, consumes few resources, scales well with the number of clients, and interacts well with other TCP flows. We also illustrate, through macrobenchmarks, the types of new services that the Trickles approach enables.

We have implemented the Trickles protocol stack in the Linux 2.4.26 kernel. Our Linux protocol stack implements the full transport protocol, the interface and the SKIP and parallel request mechanisms described earlier. The implementation consists of 15,000 total lines of code, structured as a loadable kernel module, with minimal hooks added to the base kernel. We use AES [9] for the keyed cryptographic hash function. All results include at least six data points; error bars indicate the 95% confidence interval.

All microbenchmarks in this section were performed on an isolated Gigabit Ethernet using 1.7GHz Pentium 4's, with 512 MB RAM, and Intel e1000 gigabit network interfaces. To test the network layer in isolation, we served all content from memory rather than disk.

6.1 Microbenchmarks

In this section, we use a high-performance server microbenchmark to examine the throughput, scaling, and TCP-friendliness properties of Trickles.

Throughput

We tested throughput using a point-to-point topology with a single server node placed behind a 100 Mb/sec bottleneck link. Varying numbers of simultaneous client instances (distributed across two real CPUs) repeatedly fetched a 500 kB file from the server. A fresh connection was established for each request.

Figure 9 shows that the aggregate throughput achieved by Trickles is within 10% of TCP at all client counts. Regular TCP consumes memory separately for each connection to buffer outgoing data until it is acknowledged. Beyond 6000 clients, TCP exhausts memory, forcing the kernel to kill the server process. In contrast, the Trickles kernel does not retain outgoing data, and recomputes lost packets as necessary from the original source. Consequently, it does not suffer from a memory bottleneck.

With Trickles, a client fetching small objects will achieve significant performance improvements because of the reduction in the number of control packets (Figure 10). Trickles requires fewer packets for connection setup than TCP. Trickles processes data embedded in SYN packets into output continuations without holding state, and can send an immediate response; to avoid creating a DoS amplification vulnerability, the server should not respond with more data than it received. In contrast, TCP must save or reject SYN data; because holding state increases vulnerability to SYN flooding, most TCP stacks reject SYN data. Unlike TCP, Trickles does not require FIN packets to clean up server-side state. The combination of SYN data and lower connection overhead improves small file transfer throughput for Trickles, with a corresponding improvement in transfer latency.

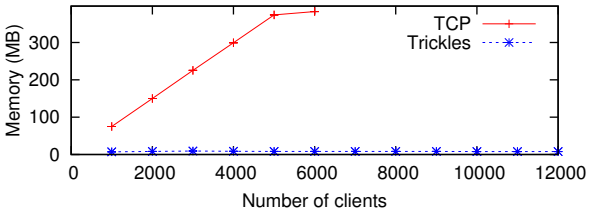


Figure 11: Memory utilization. Includes socket structures, socket buffers, and shared event queue.

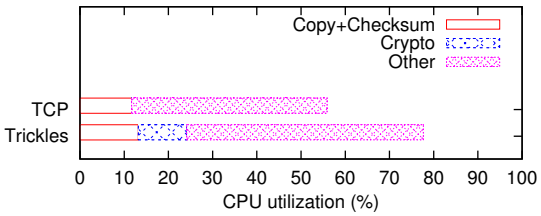


Figure 12: Server-side CPU overhead on a 1 Gb/s link.

Memory and CPU utilization

We next examine the memory and CPU utilization of the Trickles protocol. For this experiment, we eliminated the bottleneck link in the network and connected the clients to the server through the full 1Gb/sec link to pose a worst-case scenario.

Not surprisingly, Trickles consistently achieves better memory utilization than TCP (Figure 11). TCP memory utilization increases linearly with the number of clients, while statelessness enables Trickles to uses a constant, small amount of memory.

Reduced memory consumption in the network layer can improve system performance for a variety of applications. In web server installations, persistent, pipelined HTTP connections are known to reduce download latencies, though they pose a risk because increased connection duration can increase the number of simultaneous connections. Consequently, many websites disable persistent connections to the detriment of their users. Trickles can achieve the benefits of persistent connections without suffering from scalability problems. The low memory requirement of Trickles also enables small devices with restricted amounts of memory to support large numbers of connections. Finally, Trickles's smaller memory footprint provides more space for caching, benefiting all connections.

Figure 12 shows a breakdown of the CPU overhead for Trickles and TCP on a 1 Gb/s link when Trickles is reconstructing its state for every packet (i.e. soft-state caching is turned off). Not surprisingly, Trickles has higher CPU utilization than TCP, since it verifies and recomputes state that it does not keep locally. The overhead is evenly split between the cryptographic operations required for verification and the packet processing required to simulate the TCP engine. While the Trickles CPU overhead is higher, it does not pose a server bottleneck even at gigabit speeds.

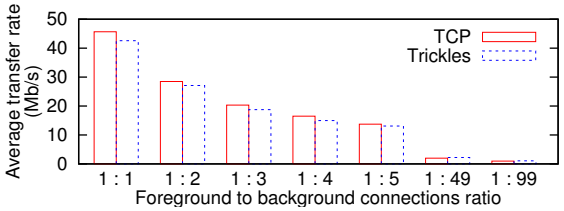


Figure 13: Interaction of Trickles and TCP.

Interaction with TCP flows

New transport protocols must not adversely affect existing flows on the Internet. Trickles is designed to generate similar packet-level behavior to TCP, and should therefore achieve similar performance as TCP under similar conditions. To confirm this, we measured the bandwidth achieved by Trickles in the presence of background flows. We constructed a dumbbell topology with two servers on the same side of a 100 Mb bottleneck link, and two clients on the other side. The remaining links from the servers and clients to their respective bottleneck routers operated at 1000 Mb. Each server was paired with one client, with connections occurring only within each server/client pair. One pair generated a single "foreground" TCP or Trickles flow. The other pair generated a variable number of background TCP flows. We compared the throughput achieved by the foreground flow for Trickles and TCP, versus varying numbers of background connections (Figure 13). In all cases, Trickles performance was similar to that of TCP.

Continuation optimizations

The SKIP and parallel continuation request mechanisms allow Trickles to efficiently support pipelined transfers, enhancing protocol performance over wide area networks. We verified their effectiveness over WAN conditions by using nistnet [7] to introduce artificial delays on a point-to-point, 100 Mb link. The single client maintained 10 outstanding pipelined requests, and the server sent advanced SKIP notifications when 50% of the file was transmitted.

We compared the performance of TCP and Trickles for pipelined connections over a point-to-point link with 10ms RTT. The file size was 250kB. This object size ensures that the link can be filled, independent of the continuation request mechanism. Trickles achieves 86 Mb/s, and TCP 91 Mb/s. Thus, with SKIP hints Trickles achieves performance similar to that of TCP.

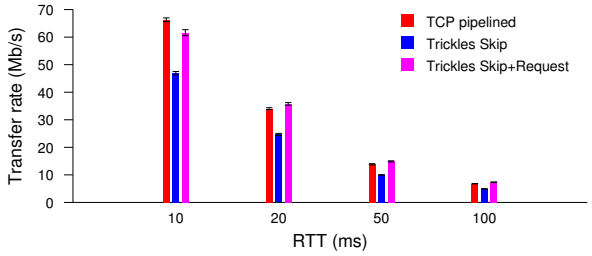


Figure 14: Throughput comparison of pipelined transfers with 20 kB objects, smaller than the bandwidth-delay product.

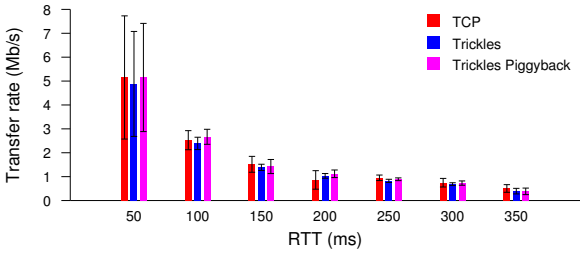


Figure 15: Trickles and TCP PlanetLab throughput.

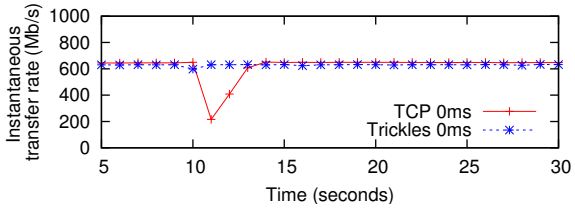


Figure 16: Failover behavior. Disconnection occurs at $t = 10$ seconds.

We also verified that issuing continuation requests in parallel improves performance. We added the `msk_request()` interface that takes application-specified data and reliably transmits the data to the server for conversion into an output continuation. These requests are non-blocking, and multiple such requests can be pending at any time. In Figure 14, the object sizes are small, so a Trickles client using SKIP with the sockets interface cannot receive output continuations quickly enough to fill the link. The Trickles client supporting parallel requests can receive continuations more frequently, resulting in performance comparable to TCP.

Summary

Compared to TCP, Trickles achieves similar or better throughput and scales asymptotically better in terms of memory. It is also TCP-friendly. Trickles incurs a significant CPU utilization overhead versus baseline TCP, but this additional CPU utilization does not pose a performance bottleneck even at gigabit speeds. The continuation management mechanisms allow Trickles to achieve performance comparable to TCP over a variety of simulated network delays and with both pipelined and non-pipelined connections.

6.2 Macrobenchmarks

The stateless Trickles protocol, and the new event-driven Trickles interface, enable a new class of stateless services. We examine three such services, and we also evaluate Trickles under real-world network loss and delay.

PlanetLab measurements

We validated Trickles under real Internet conditions using PlanetLab [5]. We ran a variant of the throughput experiment in which both the server and the client were located in our local cluster, but with all traffic between the two nodes redirected (bounced) through a single PlanetLab node m. Packets are first sent from the source node to m, then from m to the destination node. Thus, packets incur twice the underlying RTT to PlanetLab.

Figure 15 summarizes the average throughput for a 160kB file. PlanetLab nodes are grouped into 50 ms bins by the RTT measured by the endpoints. Trickles achieves similar performance to TCP under comparable network conditions.

Instantaneous failover

Trickles enables connections to fail over from a failed server to a live backup simply through a network-level redirection. If network conditions do not change significantly during the failover to invalidate the protocol parameters captured in the continuation, a server replica can resume packet processing transparently and seamlessly. In contrast, TCP recovery from server failure fundamentally requires several out of band operations. TCP needs to detect the disconnection, re-establish the connection with another server, and then ramp back up to the original data rate.

We compared Trickles and TCP failover on a 1000 Mb single server/single client connection. At 10 seconds, the server application is killed and immediately restarted. Figure 16 contains a trace illustrating the recovery of Trickles and TCP. Since transient server failures are equivalent to packet loss at the network level, Trickles flows can recover quickly and transparently using fast recovery or slow start. The explicit recovery steps needed by TCP increases its recovery time.

Packet-level load balancing

Trickles requests are self-describing, and hence can be processed by any server machine. This allows the network to freely dispatch request packets to any server. With TCP, network level redirection must ensure that

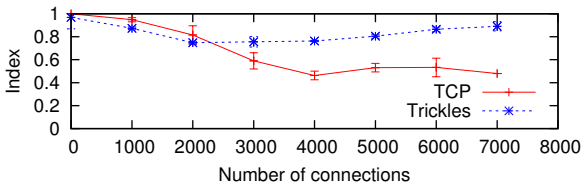


Figure 17: Jain's fairness index in load balancing cluster with two servers and two clients. Allocation is fair when each client receives the same number of bytes.

packets from a particular flow are always delivered to the same server. Hence, Trickles allows load balancing at packet granularity, whereas TCP allows load balancing only at connection granularity.

Packet-level granularity improves bandwidth allocation. We used an IP layer packet sprayer to implement a clustered web server with two servers and two clients. The IP packet sprayer uses NAT to present a single external server IP to the clients. In the test topology, the clients, servers, and packet sprayer are connected to a single Ethernet switch. The servers are connected to the switch at 100 Mb to introduce a single bottleneck on the server–client path.

TCP and Trickles tests used different load balancing algorithms. TCP connections were assigned to servers using the popular "least connections" heuristic, which permanently assigns new TCP connections to the node with the least number of connections at arrival time. Trickles connections were processed using a per-packet algorithm that dispatched packets on a round-robin schedule.

Figure 17 compares the Jain's fairness index[14] of the total throughput versus the uniform allocation. For most data points, Trickles more closely matches the uniform distribution than TCP does.

Dynamic content

Loss recovery in a stateless system may require the recomputation of past data; this is more challenging for dynamic content. To demonstrate the generality of stateless servers, we implemented a watermarking media server that modifies standard media files to custom versions containing a client-specific watermark. Such servers are relevant for DRM media distribution systems, where content providers may apply client-specific transforms to digital media before transmission. Client customization inherently prevents multiple simultaneous downloads of the same object from sharing socket buffers, thus increasing the memory footprint of the network stack.

We built a JPEG watermarking application that provides useful insights into continuation encoding for stateless operation. JPEG relies on Huffman coding of image data, which requires a non-trivial continuation structure. The exact bit position of a particular symbol after Huffman coding is not purely stateless, as it is dependent on the bit position of the previous symbols.

In our Trickles-based implementation of such a server, the output continuation records the bit alignments of encoded JPEG coding units at regular intervals. When generating output continuations, the server runs the watermarking algorithm to determine these bit positions, and discards the actual data. While processing a request, the server consults the bit positions in the output continuation for the proper bit alignment to use for the response.

7 Related Work

Previous work has noted the enhanced scalability and security properties of stateless protocols and algorithms. Aura et al. [2] developed a general framework for converting stateful protocols to stateless protocols, and applied this to authentication protocols, and noted denial-of-service resilience and potential for anycast applications as benefits of stateless protocols. Trickles deals with the more general problem of streaming data, provides a high performance stateless congestion control algorithm. Stateless Core Routing (SCORE) [22] redistributes state in routing algorithms to improve scalability. Rather than placing state at the core routers, where holding state is expensive and often infeasible, SCORE moves the state to the edge of the network.

Continuations are used in several existing systems. SYN cookies are a classic modification to TCP that uses a simple continuation to eliminate per-connection state during connection setup [6, 27]. NFS directory cookies [25] are application continuations.

Continuations for Internet services have been explored at a coarser granularity than in Trickles. Session-based mobility [21] adds continuations at the application layer to support migration and load balancing. Service Continuations [23, 24] record state snapshots, and move these to new servers during migration. In these systems, continuations are large and used infrequently in explicit migration operations controlled by connection endpoints. Trickles provides continuations at packet level, enabling new functionality within the network infrastructure.

Receiver-driven protocols [12, 13] provide clients with more control over congestion control. Since congestion often occurs near clients, and is consequently more readily detectable by the client, such systems can adapt to congestion more quickly. Trickles contributes a secure, light-weight congestion control algorithm that enforces strong guarantees on receiver behavior.

Several kernel interfaces address the memory and event-processing overhead of network stacks. IO-lite [18] reduces memory overhead by enabling buffer sharing between different connections and the filesystem. Dynamic buffer tuning [20] allocates socket buffer space

to connections where it is most needed. Event interfaces such as epoll(), kqueue(), and others [16, 4] provide efficient mechanisms for multiplexing events from different connections.

8 Conclusions and future work

Trickles demonstrates that it is possible to build a completely stateless network stack that offers many of the desirable properties of TCP; namely, efficient, reliable transmission of data streams between two endpoints. As a result, the stateless side of a Trickles connection can offer good performance with a very small memory footprint. Statelessness in Trickles extends all the way into applications: the server-side API enables servers to export their state to the client through a user continuation mechanism. Cryptographic hashes prevent untrusted clients from tampering with server state. Trickles is backwards compatible with existing TCP clients and servers, and can be adopted incrementally.

Beyond efficiency and scalability, statelessness enables new functionality that is awkward or impossible in a stateful system. Trickles enables load-balancing at packet granularity, instantaneous failover via packet redirection, and transparent connection migration. Trickles servers may be replicated, geographically distributed, and contacted through an anycast primitive, and yet provide the same semantics as a single stateful server.

Statelessness is a valuable property in many domains. The techniques used to convert TCP to a stateless protocol—for example, the methods for working around the intrinsic information propagation delays—may also have applications to other network protocols and distributed systems.

Acknowledgments

We would like to thank Paul Francis, Larry Peterson, and the anonymous reviewers for their feedback.

This work was supported by the Department of the Navy, Office of Naval Research, ONR Grant N00014-01-1-0968; and National Science Foundation grants 0208642, 0133302, and 0430161. Andrew Myers is supported by an Alfred P. Sloan Research Fellowship. Opinions, findings, conclusions, or recommendations contained in this material are those of the authors and do not necessarily reflect the views of these sponsors. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. *RFC 2581*, Apr. 1999.

[2] T. Aura and P. Nikander. Stateless Connections. In *Proceedings of the International Conference on Information and Communication Security*, pages 87–97, Beijing, China, Nov. 1997.

[3] H. Ballani and P. Francis. Towards a Deployable IP Anycast Service. In *Proceedings of the Workshop on Real, Large Distributed Systems*, San Francisco, CA, Dec. 2004.

[4] G. Banga, J. C. Mogul, and P. Druschel. A Scalable and Explicit Event Delivery Mechanism for UNIX. In *Proceedings of the USENIX Annual Technical Conference*, pages 253–265, Monterey, CA, June 1999.

[5] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating Systems Support for Planetary-Scale Network Services. In *Proceedings of the Symposium on Networked Systems Design and Implementation*, San Francisco, CA, Mar. 2004.

[6] D. Bernstein. SYN Cookies. http://cr.yp.to/syncookies.html.

[7] M. Carson and D. Santay. NIST Net. http://www-x.antd.nist.gov/nistnet.

[8] R. Chakravorty, S. Banerjee, P. Rodriguez, J. Chesterfield, and I. Pratt. Performance Optimizations for Wireless Wide-Area Networks: Comparative Study and Experimental Evaluation. In *Proceedings of MobiCom*, Philadelphia, PA, Sept. 2004.

[9] J. Daemen and V. Rijmen. AES Proposal: Rijndael, 1999.

[10] D. Ely, N. Spring, D. Wetherall, S. Savage, and T. Anderson. Robust Congestion Signaling. In *Proceedings of the International Conference on Network Protocols*, pages 332 – 341, Riverside, CA, Nov. 2001.

[11] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP / 1.1. *RFC 2616*, June 1999.

[12] R. Gupta, M. Chen, S. McCanne, and J. Walrand. A Receiver-Driven Transport Protocol for the Web. In *Proceedings of INFORMS*, San Antonio, TX, Nov. 2000.

[13] H.-Y. Hsieh, K.-H. Kim, Y. Zhu, and R. Sivakumar. A Receiver-Centric Transport Protocol for Mobile Hosts with Heterogeneous Wireless Interfaces. In *Proceedings of MobiCom*, San Diego, CA, Sept. 2003.

[14] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley and Sons, Inc., 1991.

[15] B. Krishnamurthy, J. C. Mogul, and D. M. Kristol. Key Differences between HTTP/1.0 and HTTP/1.1. In *Proceedings of the World Wide Web Conference*, Toronto, Canada, May 1999.

[16] J. Lemon. Kqueue: A Generic and Scalable Event Notification Facility. In *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, June 2001.

[17] J. Mogul, L. Brakmo, D. E. Lowell, D. Subhraveti, and J. Moore. Unveiling the Transport. *SIGCOMM Computer Communications Review*, 34(1):99–106, 2004.

[18] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, New Orleans, LA, Feb. 1999.

[19] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP Congestion Control with a Misbehaving Receiver. *SIGCOMM Computer Communications Review*, 29(5):71–78, 1999.

[20] J. Semke, J. Mahdavi, and M. Mathis. Automatic TCP Buffer Tuning. In *Proceedings of ACM SIGCOMM*, Vancouver, Canada, Aug. 1998.

[21] A. C. Snoeren. *A Session-Based Approach to Internet Mobility*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Dec. 2002.

[22] I. Stoica. *Stateless Core: A Scalable Approach for Quality of Service in the Internet*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, 2000.

[23] F. Sultan. *System Support for Service Availability, Remote Healing and Fault Tolerance Using Lazy State Propagation*. PhD thesis, Division of Computer and Information Sciences, Rutgers University, Oct. 2004.

[24] F. Sultan, A. Bohra, and L. Iftode. Service Continuations: An Operating System Mechanism for Dynamic Migration of Internet Service Sessions. In *Proceedings of the Symposium on Reliable Distributed Systems*, Florence, Italy, Oct. 2003.

[25] Sun Microsystems. NFS: Network File System Protocol Specification. *RFC 1094*, Mar. 1989.

[26] G. Wright and W. Stevens. *TCP/IP Illustrated, Volume 2*. Addison-Wesley Publishing Company, Reading, MA, Oct. 1997.

[27] A. Zúquete. Improving the Functionality of SYN Cookies. In *Proceedings of the IFIP Communications and Multimedia Security Conference*, Portoroz, Slovenia, Sept. 2002.

Designing Extensible IP Router Software

Mark Handley[†*] Eddie Kohler[‡*] Atanu Ghosh[*] Orion Hodson[*] Pavlin Radoslavov[*]
[*]*International Computer Science Institute* [†]*University College, London* [‡]*UCLA*
{*mjh, kohler, atanu, hodson, pavlin*}*@xorp.org*

ABSTRACT

Many problems with today's Internet routing infrastructure—slow BGP convergence times exacerbated by timer-based route scanners, the difficulty of evaluating new protocols—are not architectural or protocol problems, but *software* problems. Router software designers have tackled scaling challenges above all, treating extensibility and latency concerns as secondary. At this point in the Internet's evolution, however, further scaling and security issues require tackling latency and extensibility head-on.

We present the design and implementation of XORP, an IP routing software stack with strong emphases on latency, scaling, and extensibility. XORP is event-driven, and aims to respond to routing changes with minimal delay—an increasingly crucial requirement, given rising expectations for Internet reliability and convergence time. The XORP design consists of a composable framework of routing processes, each in turn composed of modular processing stages through which routes flow. Extensibility and latency concerns have influenced XORP throughout, from IPC mechanisms to process arrangements to intra-process software structure, and leading to novel designs. In this paper we discuss XORP's design and implementation, and evaluate the resulting software against our performance and extensibility goals.

1 INTRODUCTION

The Internet has been fabulously successful; previously unimagined applications frequently arise, and changing usage patterns have been accommodated with relative ease. But underneath this veneer, the low-level protocols that support the Internet have largely ossified, and stresses are beginning to show. Examples include security and convergence problems with BGP routing [18], deployment problems with multicast [10], QoS, and IPv6, and the lack of effective defense mechanisms against denial-of-service attacks. The blame for this ossification has been placed at various technical and non-technical points in the Internet architecture, from limits of layered protocol design [4] to the natural conservatism of commercial interests [9]; suggested solutions have included widespread overlay networks [23, 24] and active networking [6, 30]. But less attention has been paid to a simple, yet fundamental, underlying cause: the lack of extensible, robust, high-performance router software.

The router software market is closed: each vendor's routers will run only that vendor's software. This makes it almost impossible for researchers to experiment in real networks, or to develop proof-of-concept code that might convince network operators that there are alternatives to current practice. A lack of open router APIs additionally excludes startup companies as a channel for change.

The solution seems simple in principle: router software should have open APIs. (This somewhat resembles active networks, but we believe that a more conservative approach is more likely to see real-world deployment.) Unfortunately, extensibility can conflict with the other fundamental goals of performance and robustness, and with the sheer complexity presented by routing protocols like BGP. Relatively few software systems have robustness and security goals as stringent as those of routers, where localized instability or misconfiguration can ripple throughout the Internet [3]. Routers must also juggle hundreds of thousands of routes, which can be installed and withdrawn en masse as links go up and down. This limits the time and space available for extensions to run. Unsurprisingly, then, existing router software was not written with third-party extension in mind, so it doesn't generally include the right hooks, extension mechanisms and security boundaries.

We therefore saw the need for a new suite of router software: an integrated open-source software router platform running on commodity hardware, and viable both in research and production. The software architecture would have extensibility as a primary goal, permitting experimental protocol deployment with minimal risk to existing services. Internet researchers needing access to router software would share a common platform for experimentation, and get an obvious path to deployment for free. The loop between research and realistic real-world experimentation would eventually close, allowing innovation to take place much more freely. We have made significant progress towards building this system, which we call *XORP*, the eXtensible Open Router Platform [13].

This paper focuses on the XORP control plane: routing protocols, the Routing Information Base (RIB), net-

Orion Hodson is currently at Microsoft Research.

This material is based upon work supported by the National Science Foundation under Grant No. 0129541. Any opinions, findings, and conclusions or recommendations expressed in the material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

work management software, and related user-level programs that make up the vast majority of software on a router today. This contrasts with the forwarding plane, which processes every packet passing through the router. Prior work on component-based forwarding planes has simultaneously achieved extensibility and good performance [16, 26], but these designs, which are based on the flow of packets, don't apply directly to complex protocol processing and route wrangling. XORP's contributions, then, consist of the strategies we used to break the control plane, and individual routing protocols, into components that facilitate both extension and good performance.

For example, we treat both BGP and the RIB as networks of routing *stages*, through which routes flow. Particular stages within those networks can combine routes from different sources using various policies, or notify other processes when routes change. Router functionality is separated into many Unix processes for robustness. A flexible IPC mechanism lets modules communicate with each other independent of whether those modules are part of the same process, or even on the same machine; this allows untrusted processes to be run entirely sandboxed, or even on different machines from the forwarding engine. XORP processes are event-driven, avoiding the widely-varying delays characteristic of timer-based designs (such as those deployed in most Cisco routers). Although XORP is still young, these design choices are stable enough to have proven their worth, and to demonstrate that extensible, scalable, and robust router software is an achievable goal.

The rest of this paper is organized as follows. After discussing related work (Section 2), we describe a generic router control plane (Section 3) and an overview of XORP (Section 4). Sections 5 and 6 describe particularly relevant parts of the XORP design: the routing stages used to compose the RIB and routing protocols like BGP and our novel inter-process communication mechanism. The remaining sections discuss our security framework; present a preliminary evaluation, which shows that XORP's extensible design does not impact its performance on macro-benchmarks; and conclude.

2 Related Work

Previous work discussed XORP's requirements and high-level design strategy [13]; this paper presents specific solutions we developed to achieve those requirements. We were inspired by prior work on extensible forwarding planes, and support Click [16], one such forwarding plane, already.

Individual open-source routing protocols have long been available, including *routed* [29] for RIP, *OSPFd* [20] for OSPF, and *pimd* [14] for PIM-SM multicast routing. However, interactions between protocols can be problematic unless carefully managed. *GateD* [21] is perhaps the best known integrated routing suite, although it began as an implementation of a single routing protocol. GateD is a single process within which all routing protocols run. Such monolithic designs are fundamentally at odds with the concept of differentiated trust, whereby more experimental code can be run alongside existing services without destabilizing the whole router. MRTD [28] and BIRD [2], two other open-source IP router stacks, also use a single-process architecture. In the commercial world, Cisco IOS [7] is also a monolithic architecture; experience has shown that this significantly inhibits network operators from experimenting with Cisco's new protocol implementations.

Systems that use a multi-process architecture, permitting greater robustness, include Juniper's JunOS [15] and Cisco's most recent operating system IOS XR [8]. Unfortunately, these vendors do not make their APIs accessible to third-party developers, so we have no idea if their internal structure is well suited to extensibility. The open-source Zebra [31] and Quagga [25] stacks use multiple processes as well, but their shared inter-process API is limited in capability and may deter innovation.

Another important distinguishing factor between implementations is whether a router is *event-driven* or uses a periodic *route scanner* to resolve dependencies between routes. The scanner-based approach is simpler, but has a rather high latency before a route change actually takes effect. Cisco IOS and Zebra both use route scanners, with (as we demonstrate) a significant latency cost; MRTD and BIRD are event-driven, but this is easier given a single monolithic process. In XORP, the decision that everything is event-driven is fundamental and has been reflected in the design and implementation of all protocols, and of the IPC mechanism.

3 Control Plane Functional Overview

The vast majority of the software on a router is control-plane software: routing protocols, the Routing Information Base (RIB), firewall management, command-line interface, and network management—and, on modern routers, much else, including address management and "middlebox" functionality. Figure 1 shows a basic functional breakdown of the most common software on a router. The diagram's relationships correspond to those in XORP and, with small changes, those in any router. The rest of this section explores those relationships further.

The unicast routing protocols (BGP, RIP, OSPF, and IS-IS) are clearly functionally separate, and most routers only run a subset of these. However, as we will see later, the coupling between routing protocols is fairly complex. The arrows on the diagram illustrate the major flows of routing information, but other flows also exist.

The Routing Information Base (RIB) serves as the plumbing between routing protocols. Protocols such as

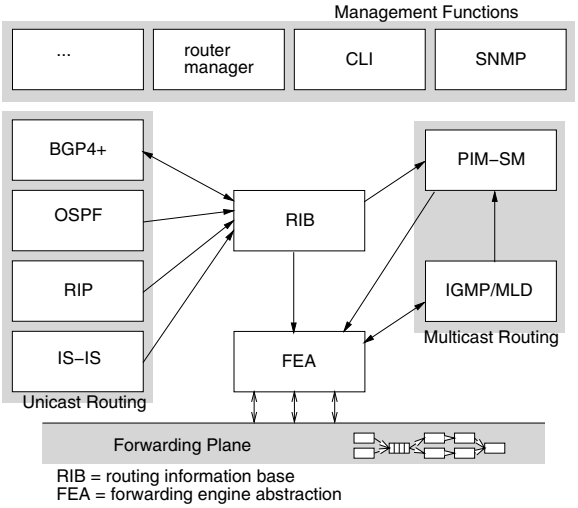


FIGURE 1—Typical router control plane functions

RIP and OSPF receive routing information from remote routers, process it to discover feasible routes, and send these routes to the RIB. As multiple protocols can supply different routes to the same destination subnet, the RIB must arbitrate between alternatives.

BGP has a more complex relationship with the RIB. Incoming IBGP routes normally indicate a *nexthop* router for a destination, rather than an immediate neighbor. If there are multiple IBGP routes to the same subnet, BGP will typically need to know the routing metrics for each choice so as to decide which route has the nearest exit (so-called "hot potato" routing). Thus, BGP must examine the routing information supplied to the RIB by other routing protocols to make its own routing decisions.

A key instrument of routing policy is the process of route redistribution, where routes from one routing protocol that match certain policy filters are redistributed into another routing protocol for advertisement to other routers. The RIB, as the one part of the system that sees everyone's routes, is central to this process.

The RIB is thus crucial to the correct functioning of a router, and should be extended only with care. Routing protocols may come and go, but the RIB should ideally be general enough to cope with them all; or failing that, it should support small, targeted extensions that are easily checked for correctness.

The Forwarding Engine Abstraction (FEA) provides a stable API for communicating with a forwarding engine or engines. In principle, its role is syntactic, and many single-platform routers leave it out, communicating with the forwarding plane directly.

PIM-SM (Protocol Independent Multicast—Sparse Mode [12]) and IGMP provide multicast routing functionality, with PIM performing the actual routing and IGMP informing PIM of the existence of local receivers. PIM contributes routes not to the RIB, but directly via the FEA to the forwarding engine. Thus, the FEA's interface is important for more than just the RIB. However, PIM does use the RIB's routing information to decide on the reverse path back to a multicast source.

The "Router Manager" holds the router configuration and starts, configures, and stops protocols and other router functionality. It hides the router's internal structure from the user, providing operators with unified management interfaces for examination and reconfiguration.

Our goal is a router control plane that provides all this functionality, including all the most widely used routing protocols, in a way that encourages extensibility. At this point, we do *not* automatically protect operators from malicious extensions or experimental code. Instead, our software architecture aims to *minimize extension footprint*, making it feasible for operators to check the code themselves. This requires a fundamental design shift from the monolithic, closely-coupled designs currently prevalent. In Section 7 we will discuss in more detail our current and future plans for XORP's security framework.

4 XORP OVERVIEW

The XORP control plane implements this functionality diagram as a set of communicating processes. Each routing protocol and management function is implemented by a separate process, as are the RIB and the FEA. Processes communicate with one another using an extensible IPC mechanism called XORP Resource Locators, or XRLs. This blurs the distinction between intra- and inter-process calls, and will even support transparent communication with non-XORP processes. The one important process not represented on the diagram is the *Finder*, which acts as a broker for IPC requests; see Section 6.2. (XORP 1.0 supports BGP and RIP; support for OSPF and IS-IS is under development.)

This multi-process design limits the coupling between components; misbehaving code, such as an experimental routing protocol, cannot directly corrupt the memory of another process. Performance is a potential downside, due to frequent IPCs; to address it, we implemented various ways to safely cache IPC results such as routes (Section 5.2.1). The multi-process approach also serves to decouple development for different functions, and encourages the development of stable APIs. Protocols such BGP and RIP are not special in the XORP design—they use APIs equally available to all. Thus, we have confidence that those APIs would prove sufficient, or nearly so, for most experimental routing protocols developed in the future.[1]

We chose to implement XORP primarily in C++, because of its object orientation and good performance. Realistic alternatives would have been C and Java. When we started implementing XORP, the choice was not com-

pletely clear cut, but we've become increasingly satisfied; for example, extensive use of C++ templates allows common source code to be used for both IPv4 and IPv6, with the compiler generating efficient implementations for both.

Each XORP process adopts a single-threaded event-driven programming model. An application such as a routing protocol, where events affecting common data come from many sources simultaneously, would likely have high locking overhead; but, more importantly, our experience is that it is very hard for new programmers to understand a multi-threaded design to the point of being able to extend it safely. Of course, threaded programs could integrate with XORP via IPC.

The core of XORP's event-driven programming model is a traditional `select`-based *event loop* based on the SFS toolkit [19]. Events are generated by timers and file descriptors; callbacks are dispatched whenever an event occurs. Callbacks are type-safe C++ functors, and allow for the currying of additional arguments at creation time.

When an event occurs, we attempt to process that event to completion, including figuring out all inter-process dependencies. For example, a RIP route may be used to resolve the nexthop in a BGP route; so a RIP route change must immediately notify BGP, which must then figure out all the BGP routes that might change as a result. Calculating these dependencies quickly and efficiently is difficult, introducing strong pressure toward a periodic route scanner design. Unfortunately, periodic scanning introduces variable latency and can lead to increased load bursts, which can affect forwarding performance. Since low-delay route convergence is becoming critical to ISPs, we believe that future routing implementations must be event-driven.

Even in an event-driven router, some tasks cannot be processed to completion in one step. For example, a router with a full BGP table may receive well over 100,000 routes from a single peer. If that peering goes down, all these routes need to be withdrawn from all other peers. This can't happen instantaneously, but a flapping peer should not prevent or unduly delay the processing of BGP updates from other peers. Therefore, XORP supports background tasks, implemented using our timer handler, which run only when no events are being processed. These background tasks are essentially cooperative threads: they divide processing up into small slices, and voluntarily return execution to the process's main event loop from time to time until they complete.

We intend for XORP to run on almost any modern operating system. We initially provide support, including FEA support, for FreeBSD and Linux, and for FreeBSD and Linux running Click as a forwarding path. Windows support is under development.

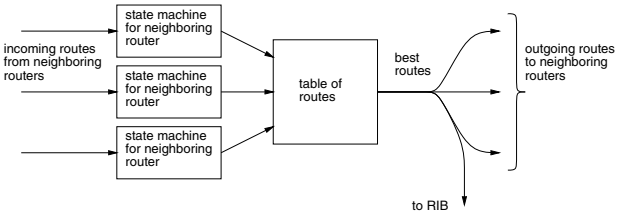


FIGURE 2—Abstract routing protocol

5 ROUTING TABLE STAGES

From the general process structure of the XORP control plane, we now turn to modularity and extensibility *within* single processes, and particularly to the ways we divide routing table processing into *stages* in BGP and the RIB. This modularization makes route dataflow transparent, simplifies the implementation of individual stages, clarifies overall organization and protocol interdependencies, and facilitates extension.

At a very high level, the abstract model in Figure 2 can represent routing protocols such as RIP or BGP. (Link-state protocols differ slightly since they distribute *all* routing information to their neighbors, rather than just the best routes.) Note that packet formats and state machines are largely separate from route processing, and that all the real magic—route selection, policy filtering, and so forth—happens within the table of routes. Thus, from a software structuring point of view, the interesting part is the table of routes.

Unfortunately, BGP and other modern routing protocols are big and complicated, with many extensions and features, and it is very hard to understand all the interactions, timing relationships, locking, and interdependencies that they impose on the route table. For instance, as we mentioned, BGP relies on information from intra-domain routing protocols (IGPs) to decide whether the nexthop in a BGP route is actually reachable and what the metric is to that nexthop router. Despite these dependencies, BGP must scale well to large numbers of routes and large numbers of peers. Thus, typical router implementations put all routes in the same memory space as BGP, so that BGP can directly see all the information relevant to it. BGP then periodically walks this jumbo routing table to figure out which routes win, based on IGP routing information. This structure is illustrated in Figure 3. While we don't *know* how Cisco implements BGP, we can infer from clues from Cisco's command line interface and manuals that it probably works something like this.

Unfortunately, this structure makes it very hard to separate functionality in such a way that future programmers can see how the pieces interact or where it is safe to make changes. Without good structure we believe that it will be impossible for future programmers to extend our

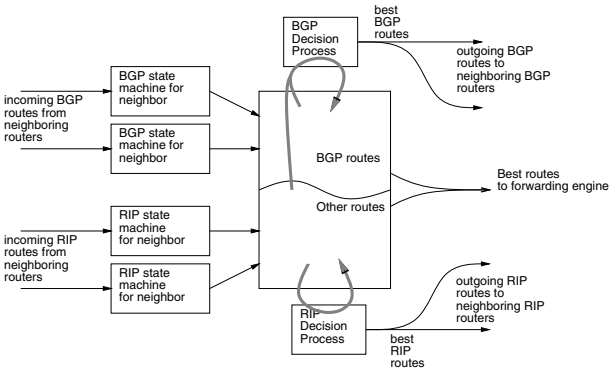


FIGURE 3—Closely-coupled routing architecture

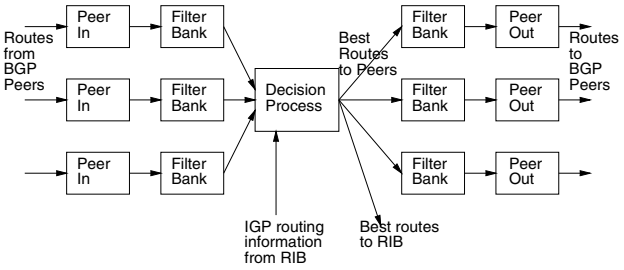


FIGURE 4—Staged BGP architecture

software without compromising its stability.

Our challenge is to implement BGP and the RIB in a more decoupled manner that clarifies the interactions between modules.

5.1 BGP Stages

The mechanism we chose is the clear one of data flow. Rather than a single, shared, passive table that stores information and annotations, we implement routing tables as dynamic processes through which routes flow. There is no single routing table object, but rather a network of pluggable routing *stages*, each implementing the same interface. Together, the network stages combine to implement a routing table abstraction. Although unusual—to our knowledge, XORP is the only router using this design—stages turn out to be a natural model for routing tables. They clarify protocol interactions, simplify the movement of large numbers of routes, allow extension, ease unit testing, and localize complex data structure manipulations to a few objects (namely trees and iterators; see Section 5.3). The cost is a small performance penalty and slightly greater memory usage, due to some duplication between stages. To quantify this, a XORP router holding a full backbone routing table of about 150,000 routes requires about 120 MB for BGP and 60 MB for the RIB, which is simply not a problem on any recent hardware. The rest of this section develops this stage design much as we developed it in practice.

To a first approximation, BGP can be modeled as the pipeline architecture, shown in Figure 4. Routes come in from a specific BGP peer and progress through an incoming filter bank into the decision process. The best routes then proceed down additional pipelines, one for each peering, through an outgoing filter bank and then on to the relevant peer router. Each stage in the pipeline receives routes from upstream and passes them downstream, sometimes modifying or filtering them along the way. Thus, stages have essentially the same API, and are indifferent to their surroundings: new stages can be added to the pipeline without disturbing their neighbors, and their interactions with the rest of BGP are constrained by the stage API.

The next issue to resolve is where the routes are actually stored. When a new route to a destination arrives, BGP must compare it against *all* alternative routes to that destination (not just the previous winner), which dictates that all alternative routes need to be stored. The natural place might seem to be the Decision Process stage; but this would complicate the implementation of filter banks: Filters can be changed by the user, after which we need to re-run the filters and re-evaluate which route won. Thus, we only store the original versions of routes, in the Peer In stages. This in turn means that the Decision Process must be able to look up alternative routes via calls *upstream* through the pipeline.

The basic interface for a stage is therefore:

- add_route: A preceding stage is sending a new route to this stage. Typically the route will be dropped, modified, or passed downstream to the next stage unchanged.
- delete_route: A preceding stage is sending a delete message for an old route to this stage. The deletion should be dropped, modified, or passed downstream to the next stage unchanged.
- lookup_route: A later stage is asking this stage to look up a route for a destination subnet. If the stage cannot answer the request itself, it should pass the request upstream to the preceding stage.

These messages can pass up and down the pipeline, with the constraint that messages must be consistent. There are two consistency rules: (1) Any delete_route message must correspond to a previous add_route message; and (2) the result of a lookup_route should be consistent with previous add_route and delete_route messages sent downstream. These rules lessen the stage implementation burden. A stage can assume that upstream stages are consistent, and need only *preserve* consistency for downstream stages.

For extra protection, a BGP pipeline could include stages that *enforced* consistency around possibly-erro-

neous experimental extensions, but so far we have not needed to do this. Instead, we have developed an extra consistency checking stage for debugging purposes. This cache stage, just after the outgoing filter bank in the output pipeline to each peer, has helped us discover many subtle bugs that would otherwise have gone undetected. While not intended for normal production use, this stage could aid with debugging if a consistency error is suspected.

5.1.1 Decomposing the Decision Process

The Decision Process in this pipeline is rather complex: in addition to deciding which route wins, it must get nexthop resolvability and metric information from the RIB, and fan out routing information to the output peer pipeline branches and to the RIB. This coupling of functionality is undesirable both because it complicates the stage, and because there are no obvious extension points within such a macro-stage. XORP thus further decomposes the Decision Process into Nexthop Resolvers, a simple Decision Process, and a Fanout Queue, as shown in Figure 5.

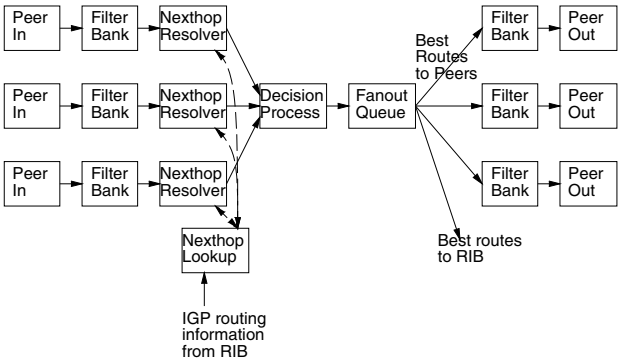


FIGURE 5—Revised staged BGP architecture

The Fanout Queue, which duplicates routes for each peer and for the RIB, is in practice complicated by the need to send routes to slow peers. Routes can be received from one peer faster than we can transit them via BGP to other peers. If we queued updates in the *n* Peer Out stages, we could potentially require a large amount of memory for all *n* queues. Since the outgoing filter banks modify routes in different ways for different peers, the best place to queue changes is in the fanout stage, after the routes have been chosen but before they have been specialized. The Fanout Queue module then maintains a single route change queue, with *n* readers (one for each peer) referencing it.

The Nexthop Resolver stages talk asynchronously to the RIB to discover metrics to the nexthops in BGP's routes. As replies arrive, it annotates routes in add_route and lookup_route messages with the relevant IGP metrics. Routes are held in a queue until the relevant nexthop metrics are received; this avoids the need for the Decision Process to wait on asynchronous operations.

5.1.2 Dynamic Stages

The BGP process's stages are *dynamic*, not static; new stages can be added and removed as the router runs. We made use of this capability in a surprising way when we needed to deal with route deletions due to peer failure. When a peering goes down, all the routes received by this peer must be deleted. However, the deletion of more than 100,000 routes takes too long to be done in a single event handler. This needs to be divided up into slices of work, and handled as a background task. But this leads to a further problem: a peering can come up and go down in rapid succession, before the previous background task has completed.

To solve this problem, when a peering goes down we create a new dynamic *deletion stage*, and plumb it in directly after the Peer In stage (Figure 6).

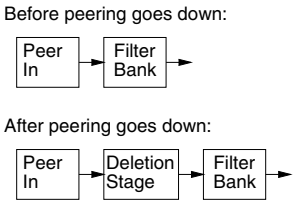


FIGURE 6—Dynamic deletion stages in BGP

The route table from the Peer In is handed to the deletion stage, and a new, empty route table is created in the Peer In. The deletion stage ensures consistency while gradually deleting all the old routes in the background; simultaneously, the Peer In—and thus BGP as a whole—is immediately ready for the peering to come back up. The Peer In doesn't know or care if background deletion is taking place downstream. Of course, the deletion stage must still ensure consistency, so if it receives an add_route message from the Peer In that refers to a prefix that it holds but has not yet got around to deleting, then first it sends a delete_route downstream for the old route, and then it sends the add_route for the new route. This has the nice side effect of ensuring that if the peering flaps many times in rapid succession, each route is held in at most one deletion stage. Similarly, routes not yet deleted will still be returned by lookup_route until after the deletion stage has sent a delete_route message downstream. In this way none of the downstream stages even know that a background deletion process is occurring—all they see are consistent messages. Even the deletion stage has no knowledge of other deletion stages; if the peering bounces multiple times, multiple dynamic deletion stages will be added, one for each time the peer-

ing goes down. They will unplumb and delete themselves when their tasks are complete.

We use the ability to add dynamic stages for many background tasks, such as when routing policy filters are changed by the operator and many routes need to be refiltered and reevaluated. The staged routing table design supported late addition of this kind of complex functionality with minimal impact on other code.

5.2 RIB Stages

Other XORP routing processes also use variants of this staged design. For example, Figure 7 shows the basic structure of the XORP RIB process. Routes come into the RIB from multiple routing protocols, which play a similar role to BGP's peers. When multiple routes are available to the same destination from different protocols, the RIB must decide which one to use for forwarding. As with BGP, routes are stored only in the origin stages, and similar add_route, delete_route and lookup_route messages traverse between the stages.

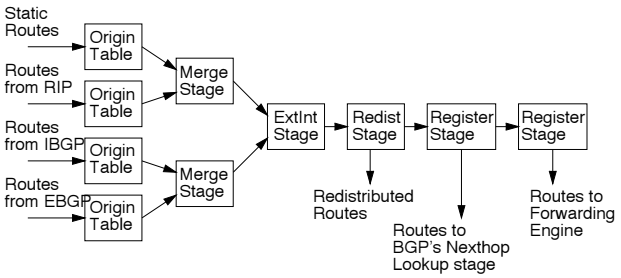


FIGURE 7—Staged RIB architecture

Unlike with BGP, the decision process in the RIB is distributed as pairwise decisions between Merge Stages, which combine route tables with conflicts based on a preference order, and an ExtInt Stage, which composes a set of external routes with a set of internal routes. In BGP, the decision stage needs to see all possible alternatives to make its choice; the RIB, in contrast, makes its decision purely on the basis of a single *administrative distance* metric. This single metric allows more distributed decision-making, which we prefer, since it better supports future extensions.

Dynamic stages are inserted as different watchers register themselves with the RIB. These include Redist Stages, which contain programmable policy filters to redistribute a route subset to a routing protocol, and Register Stages, which redistribute routes depending on prefix matches. This latter process, however, is slightly more complex than it might first appear.

5.2.1 Registering Interest in RIB Routes

A number of core XORP processes need to be able to track changes in routing in the RIB as they occur. For example, BGP needs to monitor routing changes that affect IP addresses listed as the nexthop router in BGP routes, and PIM-SM needs to monitor routing changes that affect routes to multicast source addresses and PIM Rendezvous-Point routers. We expect the same to be true of future extensions. This volume of registrations puts pressure on the Register Stage interface used to register and call callbacks on the RIB. In monolithic or shared-memory designs centered around a single routing table structure, a router could efficiently monitor the structure for changes, but such a design cannot be used by XORP. We need to share the minimum amount of information between the RIB and its clients, while simultaneously minimizing the number of requests handled by the RIB.

What BGP and PIM want to know about is the routing for specific IP addresses. But this list of addresses may be moderately large, and many addresses may be routed as part of the same subnet. Thus when BGP asks the RIB about a specific address, the RIB informs BGP about the address range for which the same answer applies.

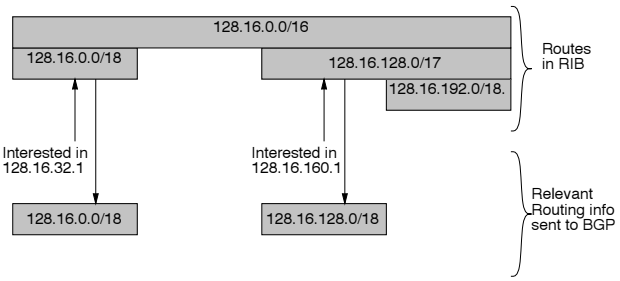


FIGURE 8—RIB interest registration

Figure 8 illustrates this process. The RIB holds routes for 128.16.0.0/16, 128.16.0.0/18, 128.16.128.0/17 and 128.16.192.0/18. If BGP asks the RIB about address 128.16.32.1, the RIB tells BGP that the matching route is 128.16.0.0/18, together with the relevant metric and nexthop router information. This address also matched 128.16.0.0/16, but only the more specific route is reported. If BGP later becomes interested in address 128.16.32.7, it does not need to ask the RIB because it already knows this address is also covered by 128.16.0.0/18.

However, if BGP asks the RIB about address 128.16.160.1, the answer is more complicated. The most specific matching route is 128.16.128.0/17, and indeed the RIB tells BGP this. But 128.16.128.0/17 is overlayed by 128.16.192.0/18, so if BGP only knew about 128.16.128.0/17 and later became interested in 128.16.192.1, it would erroneously conclude that this is also covered by 128.16.128.0/17. Instead, the RIB computes the largest enclosing subnet that is not overlayed by a more specific route (in this case 128.16.128.0/18) and tells BGP that its answer is valid for this subset of addresses only. Should

the situation change at any later stage, the RIB will send a "cache invalidated" message for the relevant subnet, and BGP can re-query the RIB to update the relevant part of its cache.

Since no largest enclosing subnet ever overlaps any other in the cached data, RIB clients like BGP can use balanced trees for fast route lookup, with attendant performance advantages.

5.3 Safe Route Iterators

Each background stage responsible for processing a large routing table, such as a BGP deletion stage, must remember its location in the relevant routing table so that it can make forward progress on each rescheduling. The XORP library includes route table *iterator* data structures that implement this functionality (as well as a Patricia Tree implementation for the routing tables themselves). Unfortunately, a route change may occur while a background task is paused, resulting in the tree node pointed to by an iterator being deleted. This would cause the iterator to hold invalid state. To avoid this problem, we use some spare bits in each route tree node to hold a reference count of the number of iterators currently pointing at this tree node. If the route tree receives a request to delete a node, the node's data is invalidated, but the node itself is not removed immediately unless the reference count is zero. It is the responsibility of the last iterator leaving a previously-deleted node to actually perform the deletion.

The internals of the implementation of route trees and iterators are not visible to the programmer using them. All the programmer needs to know is that the iterator will never become invalid while the background task is paused, reducing the feature interaction problem between background tasks and event handling tasks.

6 Inter-process Communication

Using multiple processes provides a solid basis for resource management and fault isolation, but requires the use of an inter-process communication (IPC) mechanism. Our IPC requirements were:

- to allow communication both between XORP processes and with routing applications not built using the XORP framework;
- to use multiple transports transparently, including intra-process calls, host-local IPC, and networked communication, to allow a range of tradeoffs between flexibility and performance;
- to support component namespaces for extensibility and component location for flexibility, and to provide security through per-method access control on components;
- to support asynchronous messaging, as this is a natural fit for an event-driven system; and
- to be portable, unencumbered, and lightweight.

During development we discovered an additional requirement, *scriptability*, and added it as a feature. Being able to script IPC calls is an invaluable asset during development and for regression testing. Existing messaging frameworks, such as CORBA [22] and DCOM [5], provided the concepts of components, component addressing and location, and varying degrees of support for alternative transports, but fell short elsewhere.

We therefore developed our own *XORP IPC* mechanism. The *Finder* process locates components and their methods; communication proceeds via a naturally scriptable base called *XORP Resource Locators*, or *XRLs*.

6.1 XORP Resource Locators

An XRL is essentially a method supported by a component. (Because of code reuse and modularity, most processes contain more than one component, and some components may be common to more than one process; so the unit of IPC addressing is the *component instance* rather than the process.) Each component implements an XRL *interface*, or group of related methods. When one component wishes to communicate with another, it composes an XRL and *dispatches* it. Initially a component knows only the generic component name, such as "bgp", with which it wishes to communicate. The Finder must resolve such generic XRLs into a form that specifies precisely how communication should occur. The resulting resolved XRL specifies the transport *protocol family* to be used, such as TCP, and any parameters needed for communication, such as hostname and port.

The canonical form of an XRL is textual and human-readable, and closely resembles Uniform Resource Locators (URLs [1]) from the Web. Internally XRLs are encoded more efficiently, but the textual form permits XRLs to be called from any scripting language via a simple *call_xrl* program. This is put to frequent use in all our scripts for automated testing. In textual form, a generic XRL might look like:

```
finder://bgp/bgp/1.0/set_local_as?as:u32=1777
```

And after Finder resolution:

```
stcp://192.1.2.3:16878/bgp/1.0/set_local_as?as:u32=1777
```

XRL arguments (such as "as" above, which is an Autonomous System number) are restricted to a set of core types used throughout XORP, including network addresses, numbers, strings, booleans, binary arrays, and lists of these primitives. Perhaps because our application domain is highly specialized, we have not yet needed support for more structured arguments.

As with many other IPC mechanisms, we have an interface definition language (IDL) that supports interface specification, automatic stub code generation, and basic error checking.

6.2 Components and the Finder

When a component is created within a process, it instantiates a receiving point for the relevant XRL protocol families, and then registers this with the Finder. The registration includes a component class, such as "bgp"; a unique component instance name; and whether or not the caller expects to be the sole instance of a particular component class. Also registered are each interface's supported methods and each method's supported protocol families. This allows for specialization; for example, one protocol family may be particularly optimal for a particular method.

When a component wants to dispatch an XRL, it consults the Finder for the resolved form of the XRL. In reply, it receives the resolved method name together with a list of the available protocol families and arguments to bind the protocol family to the receiver. For a networked protocol family, these would typically include the hostname, receiving port, and potentially a key. Once resolved, the dispatcher is able to instantiate a sender for the XRL and request its dispatch. XRL resolution results are cached, and these caches are updated by the Finder when entries become invalidated.

In addition to providing resolution services, the Finder also provides a component lifetime notification service. Components can request to be notified when another component class or instance starts or stops. This mechanism is used to detect component failures and component restarts.

6.3 Protocol Families

Protocol families are the mechanisms by which XRLs are transported from one component to another. Each protocol family is responsible for providing argument marshaling and unmarshaling facilities as well as the IPC mechanism itself.

Protocol family programming interfaces are small and simple to implement. In the present system, there are three protocol families for communicating between XORP components: *TCP*, *UDP*, and *intra-process*, which is for calls between components in the same process. There is also a special *Finder protocol family* permitting the Finder to be addressable through XRLs, just as any other XORP component. Finally, there exists a *kill* protocol family, which is capable of sending just one message type—a UNIX signal—to components within a host. We expect to write further specialized protocol families for communicating with non-XORP components. These will effectively act as proxies between XORP and unmodified XORP processes.

7 Security Framework

Security is a critical aspect of building a viable extensible platform. Ideally, an experimental protocol running on a XORP router could do no damage to that router, whether through poor coding or malice. We have not yet reached this ideal; this section describes how close we are.

Memory protection is of course the first step, and XORP's multi-process architecture provides this. The next step is to allow processes to be sandboxed, so they cannot access important parts of the router filesystem. XORP centralizes all configuration information in the Router Manager, so no XORP process needs to access the filesystem to load or save its configuration.

Sandboxing has limited use if a process needs to have root access to perform privileged network operations. To avoid this need for root access, the FEA is used as a relay for all network access. For example, rather than sending UDP packets directly, RIP sends and receives packets using XRL calls to the FEA. This adds a small cost to networked communication, but as routing protocols are rarely high-bandwidth, this is not a problem in practice.

This leaves XRLs as the remaining vector for damage. If a process could call any other XRL on any other process, this would be a serious problem. By default we don't accept XRLs remotely. To prevent local circumvention, at component registration time the Finder includes a 16-byte random key in the registered method name of all resolved XRLs. This prevents a process bypassing the use of the Finder for the initial XRL resolution phase, because the receiving process will reject XRLs that don't match the registered method name.

We have several plans for extending XORP's security. First, the Router Manager will pass a unique secret to each process it starts. The process will then use this secret when it resolves an XRL with the Finder. The Finder is configured with a set of XRLs that each process is allowed to call, and a set of targets that each process is allowed to communicate with. Only these permitted XRLs will be resolved; the random XRL key prevents bypassing the Finder. Thus, the damage that can be done by an errant process is limited to what can be done through its normal XRL calls. We can envisage taking this approach even further, and restricting the range of arguments that a process can use for a particular XRL method. This would require an XRL intermediary, but the flexibility of our XRL resolution mechanism makes installing such an XRL proxy rather simple. Finally, we are investigating the possibility of running different routing processes in different virtual machines under the Xen [11] virtual machine monitor, which would provide even better isolation and allow us to control even the CPU utilization of an errant process.

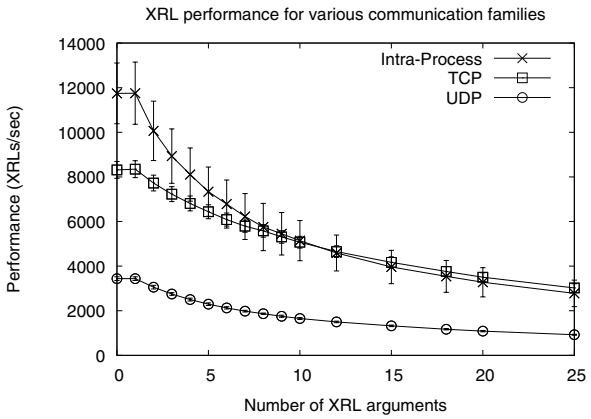


FIGURE 9—XRL performance results

8 EVALUATION

As we have discussed, the XORP design is modular, robust and extensible, but these properties will come at some cost in performance compared to more tightly coupled designs. The obvious concern is that XORP might not perform well enough for real-world use. On previous generations of hardware, this might have been true, but we will show below that it is no longer the case.

The measurements are performed on a relatively low-end PC (AMD Athlon 1100MHz) running FreeBSD 4.10. At this stage of development we have put very little effort into optimizing the code for performance, but we have paid close attention to the computation complexity of our algorithms. Nevertheless, as we show below, even without optimization the results clearly demonstrate good performance, and the advantage of our event-driven design.

8.1 XRL Performance Evaluation

One concern is that the XRL IPC mechanism might become a bottleneck in the system. To verify that it is not, the metric we are interested in is the throughput we can achieve in terms of number of XRL calls per second.

To measure the XRL rate, we send a transaction of 10000 XRLs using a pipeline size of 100 XRLs. Initially, the sender sends 100 XRLs back-to-back, and then for every XRL response received it sends a new request. The receiver measures the time between the beginning and the end of a transaction. We evaluate three communication transport mechanisms: TCP, UDP and Intra-Process direct calling where the XRL library invokes direct method calls between a sender and receiver inside the same process.[1]

[1] To allow direct comparison of Intra-Process against TCP and UDP, both sender and receiver are running within the same process. When we run the sender and receiver on two separate processes on the same host, the performance is very slightly worse.

In Figure 9 we show the average XRL rate and its standard deviation for TCP, UDP and Intra-Process transport mechanisms when we vary the number of arguments to the XRL. These results show that our IPC mechanism can easily sustain several thousands of XRLs per second on a relatively low-end PC. Not surprisingly, for a small number of XRL arguments, the Intra-Process performance is best (almost 12000 XRLs/second), but for a larger number of arguments the difference between Intra-Process and TCP disappears. It is clear from these results that our argument marshalling and unmarshalling is not terribly optimal, but despite this the results are quite respectable. In practice, most commonly used XRLs have few arguments. This result is very encouraging, because it demonstrates that typically the bottleneck in the system will be elsewhere.

The UDP performance is significantly worse because UDP was our first prototype XRL implementation, and does not pipeline requests. For normal usage, XORP currently uses TCP and does pipeline requests. UDP is included here primarily to illustrate the effect of request pipelining, even when operating locally.

8.2 Event-Driven Design Evaluation

To demonstrate the scaling properties of our event-driven design, we present some BGP-related measurements. Routing processes not under test such as PIM-SM and RIP were also running during the measurements, so the measurements represent a fairly typical real-world configuration.

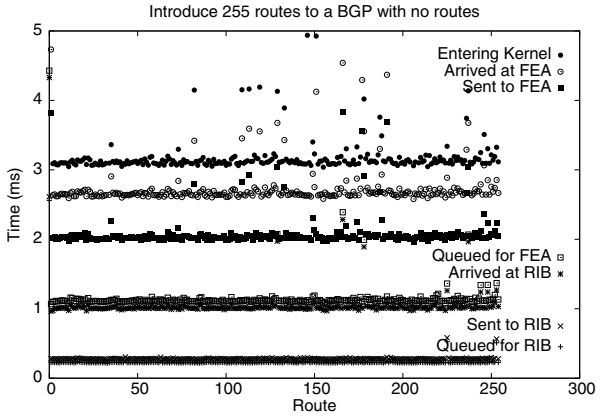
First, we perform some measurements with an empty routing table, and then with a routing table containing a full Internet backbone routing feed consisting of 146515 routes. The key metric we care about is how long it takes for a route newly received by BGP to be installed into the forwarding engine.

XORP contains a simple profiling mechanism which permits the insertion of profiling points anywhere in the code. Each profiling point is associated with a profiling variable, and these variables are configured by an external program *xorp_profiler* using XRLs. Enabling a profiling point causes a time stamped record to be stored, such as:

```
route_ribin 1097173928 664085 add 10.0.1.0/24
```

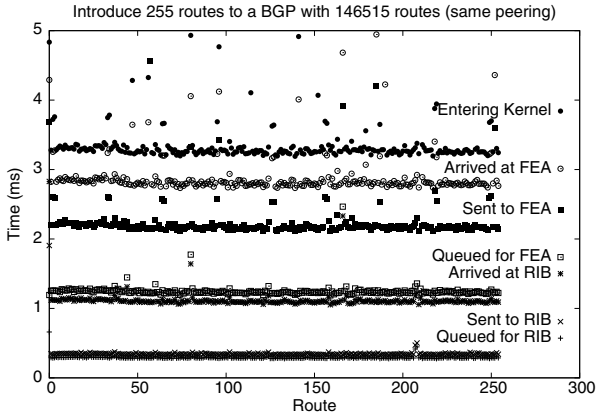
In this example we have recorded the time in seconds and microseconds at which the route "10.0.1.0/24" has been added. When this particular profiling variable is enabled, all routes that pass this point in the pipeline are logged.

If a route received by BGP wins the decision process, it will be sent to its peers and to the RIB (see Figure 1). When the route reaches the RIB, if it wins against routes from other protocols, then it is sent to the FEA. Finally,



Profile Point	Avg	SD	Min	Max
Entering BGP	-	-	-	-
Queued for transmission to the RIB	0.226	0.026	0.214	0.525
Sent to RIB	0.280	0.026	0.267	0.582
Arriving at the RIB	1.036	0.132	0.964	2.285
Queued for transmission to the FEA	1.139	0.132	1.066	2.389
Sent to the FEA	2.234	1.903	1.965	31.663
Arriving at FEA	2.876	1.918	2.580	32.443
Entering kernel	3.374	1.980	3.038	33.576

FIGURE 10—Route propagation latency (in ms), no initial routes



Profile Point	Avg	SD	Min	Max
Entering BGP	-	-	-	-
Queued for transmission to the RIB	0.289	0.015	0.275	0.447
Sent to RIB	0.344	0.015	0.329	0.502
Arriving at the RIB	1.112	0.088	1.052	2.331
Queued for transmission to the FEA	1.247	0.088	1.187	2.465
Sent to the FEA	2.362	1.571	2.108	25.910
Arriving at FEA	3.019	1.590	2.734	26.689
Entering kernel	3.632	3.394	3.191	56.422

FIGURE 11—Route propagation latency (in ms), 146515 initial routes and same peering

the FEA will unconditionally install the route in the kernel or the forwarding engine.

The following profiling points were used to measure the flow of routes:

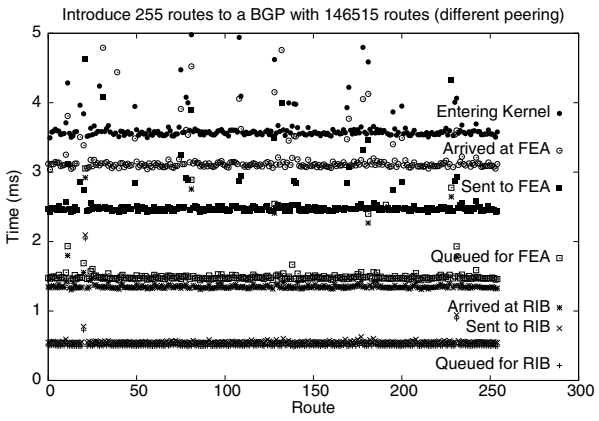
1. Entering BGP
2. Queued for transmission to the RIB
3. Sent to the RIB
4. Arriving at the RIB
5. Queued for transmission to the FEA
6. Sent to the FEA
7. Arriving at the FEA
8. Entering the kernel

One of the goals of this experiment is to demonstrate that routes introduced into a system with an empty routing table perform similarly to a system with a full BGP backbone feed of 146515 routes. In each test we introduce a new route every two seconds, wait a second, and then remove the route. The BGP protocol requires that the next hop is resolvable for a route to be used. BGP discovers if a next hop is resolvable by registering interest with the RIB. To avoid unfairly penalizing the empty routing table tests, we keep one route installed during the test to prevent additional interactions with the RIB that typically would not happen with the full routing table.

The results are shown in Figures 10–12. In the first experiment (Figure 10) BGP contained no routes other than the test route being added and deleted. In the second experiment (Figure 11) BGP contained 146515 routes and the test routes were introduced on the same peering from which the other routes were received. In the third experiment (Figure 12) BGP contained 146515 routes and the test routes were introduced on a different peering from which the other routes were received, which exercises different code-paths from the second experiment.

All the graphs have been cropped to show the most interesting region. At the tables indicate, one or two routes took as much as 90ms to reach the kernel. This appears to be due to scheduling artifacts, as FreeBSD is not a realtime operating system.

The conclusion to be drawn from these graphs is that



Profile Point	Avg	SD	Min	Max
Entering BGP	-	-	-	-
Queued for transmission to the RIB	0.508	0.100	0.483	2.039
Sent to RIB	0.563	0.101	0.536	2.094
Arriving at the RIB	1.377	0.182	1.306	2.920
Queued for transmission to the FEA	1.517	0.193	1.438	3.053
Sent to the FEA	3.149	5.943	2.419	92.391
Arriving at FEA	3.822	5.964	3.037	93.179
Entering kernel	4.417	6.278	3.494	93.662

FIGURE 12—Route propagation latency (in ms), 146515 initial routes and different peering

routing events progress to the kernel very quickly (typically within 4ms of receipt by BGP). Perhaps as importantly, the data structures we use have good performance under heavy load, therefore the latency does not significantly degrade when the router has a full routing table. The latency is mostly dominated by the delays inherent in the context switch that is necessitated by inter-process communication. We should emphasize that the XRL interface is pipelined, so performance is still good when many routes change in a short time interval.

We have argued that an event driven route processing model leads to faster convergence than the traditional route scanning approach. To verify this assertion we performed a simple experiment, shown in Figure 13. We introduced 255 routes from one BGP peer at one second intervals and recorded the time that the route appeared at another BGP peer. The experiment was performed on XORP, Cisco-4500 (IOS Version 12.1), Quagga-0.96.5, and MRTD-2.2.2a routers. It should be noted that the granularity of the measurement timer was one second.

This experiment clearly shows the consistent behavior achieved by XORP, where the delay never exceeds one second. MRTD's behavior is very similar, which is important because it illustrates that the multi-process architecture used by XORP delivers similar performance to a closely-coupled single-process architecture. The Cisco and Quagga routers exhibit the obvious symptoms of a 30-second route scanner, where all the routes received in the previous 30 seconds are processed in one batch. Fast convergence is simply not possible with such a scanner-based approach.

Teixeira et al demonstrate [27] that even route changes within an AS can be adversely affected by the delay introduced by BGP route scanners. In real ISP networks, the found delays of one to two *minutes* were common between an IGP route to a domain border router changing, and the inter-domain traffic flowing out of a domain changing its exit router. During this delay, they show that

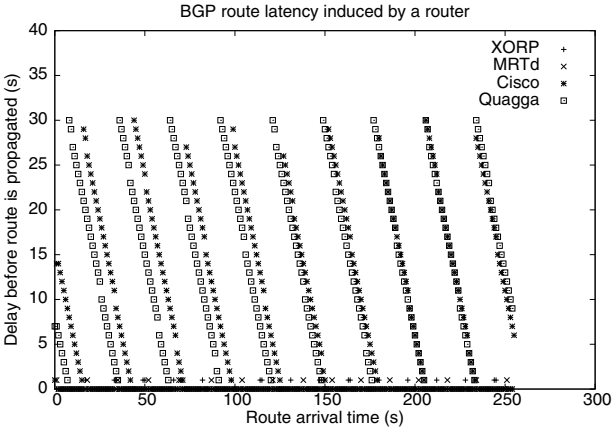


FIGURE 13—BGP route flow

transient forwarding loops can exist, or traffic may be blackholed, both of which may have significant impact on customers. Thus fast convergence is clearly of high importance to providers, and can only become more so with the increase in prominence of real-time traffic.

8.3 Extensibility Evaluation

The hardest part of our design to properly evaluate is its extensibility. Only time will tell if we really have the right modularity, flexibility, and APIs. However, we can offer a number of examples to date where extensibility has been tested.

Adding Policy to BGP

We implemented the core BGP and RIB functionality first, and only then thought about how to configure policy, which is a large part of any router functionality. Our policy framework consists of three new BGP stages and two new RIB stages, each of which supports a common simple stack language for operating on routes. The details are too lengthy for this paper, but we believe this

framework allows us to implement almost the full range of policies available on commercial routers.

The only change required to pre-existing code was the addition of a tag list to routes passed from BGP to the RIB and vice versa. Thus, our staged architecture appears to have greatly eased the addition of code that is notoriously complex in commercial vendors' products.

What we got wrong was the syntax of the command line interface (CLI) template files, described in [13], used to dynamically extend the CLI configuration language. Our original syntax was not flexible enough to allow user-friendly specification of the range of policies that we need to support. This is currently being extended.

Adding Route Flap Damping to BGP

Route flap damping was also not a part of our original BGP design. We are currently adding this functionality (ISPs demand it, even though it's a flawed mechanism), and can do so efficiently and simply by adding another stage to the BGP pipeline. The code does not impact other stages, which need not be aware that damping is occurring.

Adding a New Routing Protocol

XORP has now been used as the basis for routing research in a number of labs. One university unrelated to our group used XORP to implement an ad-hoc wireless routing protocol. In practice XORP probably did not help this team greatly, as they didn't need any of our existing routing protocols, but they did successfully implement their protocol. Their implementation required a single change to our internal APIs to allow a route to be specified by interface rather than by nexthop router, as there is no IP subnetting in an ad-hoc network.

9 Conclusions

We believe that innovation in the core protocols supporting the Internet is being seriously inhibited by the nature of the router software market. Furthermore, little long term research is being done, in part because researchers perceive insurmountable obstacles to experimentation and deployment of their ideas.

In an attempt to change the router software landscape, we have built an extensible open router software platform. We have a stable core base running, consisting of around 500,000 lines of C++ code. XORP is event-driven, giving fast routing convergence, and incorporates a multi-process design and novel inter-process communication mechanisms that aid extensibility, and allow experimental software to run alongside production software.

In this paper we have presented a range of innovative features, including a novel staged design for core protocols, and a strong internal security architecture geared around the sandboxing of untrusted components. We also presented preliminary evaluation results that confirm that our design scales well to large routing tables while maintaining low routing latency.

In the next phase we need to involve the academic community, both as early adopters, and to flesh out the long list of desirable functionality that we do not yet support. If we are successful, XORP will become a true *production-quality* platform. The road ahead will not be easy, but unless this or some other approach to enable Internet innovation is successful, the long-run consequences of ossification will be serious indeed.

Acknowledgments

We thank the other members of the XORP project, including Fred Bauer (Linux), Andrea Bittau (routing policy), Javier Cardona (SNMP support), Adam Greenhalgh (initial BGP implementation), Luigi Rizzo (fast forwarding), Bruce M. Simpson (Windows), and Marko Zec (Click integration and fast lookups). We gratefully acknowledge those who financially supported XORP, including the ICSI Center for Internet Research, Intel Corporation, the National Science Foundation (ANI-0129541), and Microsoft Corporation. We also sincerely thank Scott Shenker for believing in the work against all odds.

Notes

[1] We have some confirmation of this: a group implementing an ad-hoc routing protocol found that XORP's RIB supported their application with just one trivial interface change [17].

References

[1] T. Berners-Lee, L. Masinter, and M. McCahill. Uniform resource locators. RFC 1738, Internet Engineering Task Force, December 1994.

[2] Bird project. The BIRD Internet Routing Daemon (Web site). http://bird.network.cz/.

[3] V. J. Bono. 7007 explanation and apology, 1997. http://www.merit.edu/mail.archives/nanog/1997-04/msg00444.html.

[4] Robert Braden, Ted Faber, and Mark Handley. From protocol stack to protocol heap—Role-based architecture. In *Proc. 1st Workshop on Hot Topics in Networks (HotNets-I)*, October 2002.

[5] Nat Brown and Charlie Kindel. Distributed Component Object Model Protocol – DCOM 1.0. Online, November 1998. Expired IETF draft draft-brown-dcom-v1-spec-02.txt.

[6] Kenneth L. Calvert, James Griffioen, and Su Wen. Lightweight network support for scalable end-to-end services. In *Proc. ACM SIGCOMM 2002 Conference*, pages 265–278, August 2002.

[7] Cisco Systems. Cisco IOS software. http://www.cisco.com/public/sw-center/sw-ios.shtml.

[8] Cisco Systems. Cisco IOS XR software. http://www.cisco.com/en/US/products/ps5845/index.html.

[9] National Research Council. *Looking Over the Fence at Networks*. National Academy Press, 2001.

[10] Christophe Diot, Brian Neil Levine, Bryan Lyles, Hassan Kassem, and Doug Balensiefen. Deployment issues for the ip multicast service and architecture. *IEEE Network*, January/February 2000.

[11] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proc. 18th ACM Symposium on Operating Systems Principles*, October 2003.

[12] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei. Protocol Independent Multicast—Sparse Mode (PIM-SM): Protocol specification. RFC 2362, Internet Engineering Task Force, June 1998. ftp://ftp.ietf.org/rfc/rfc2362.txt.

[13] M. Handley, O. Hodson, and E. Kohler. XORP: An open platform for network research. In *Proc. 1st Workshop on Hot Topics in Networks (HotNets-I)*, October 2002.

[14] A. Helmy. Protocol indendent multicast-sparse mode (pim-sm) implementation document, 1996. http://netweb.usc.edu/pim/pimd/docs/.

[15] Juniper Networks. JunOS software. http://www.juniper.net/products/junos/.

[16] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Trans. on Computer Systems*, 18(3):263–297, August 2000.

[17] Thomas Kunz. Implementing bcast (implementation report). http://www.sce.carleton.ca/wmc/code.html, March 2004.

[18] Zhuoqing Morley Mao, Ramesh Govindan, George Varghese, and Randy H. Katz. Route flap damping exacerbates Internet routing convergence. In *Proc. ACM SIGCOMM 2002 Conference*, August 2002.

[19] David Mazières. A toolkit for user-level file systems. In *Proc. USENIX 2001 Annual Technical Conference*, pages 261–274, June 2001.

[20] John Moy. *OSPF Complete Implementation*. Addison-Wesley, December 2000.

[21] NextHop Technologies. GateD releases (Web site). http://www.gated.org/.

[22] Object Management Group. Common Object Request Broker Architecture Specification 3.0.3, March 2004. http://www.omg.org/cgi-bin/doc?formal/04-03-12.

[23] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A blueprint for introducing disruptive technology into the Internet. In *Proc. 1st Workshop on Hot Topics in Networks (HotNets-I)*, October 2002.

[24] Larry Peterson, Scott Shenker, and Jonathan Turner. Overcoming the internet impasse through virtualization. In *Proc. 3rd Workshop on Hot Topics in Networks (HotNets-III)*, November 2004.

[25] Quagga project. Quagga Routing Suite (Web site). http://www.quagga.net/.

[26] Tammo Spalink, Scott Karlin, Larry Peterson, and Yitzchak Gottlieb. Building a robust software-based router using network processors. In *Proc. 18th ACM Symposium on Operating Systems Principles*, pages 216–229, October 2001.

[27] R. Teixeira, A. Shaikh, T. Griffin, and J. Rexford. Dynamics of hot-potato routing in ip networks. In *Proc. 2004 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 2004.

[28] University of Michigan and Merit Network. MRT: Multi-threaded Routing Toolkit (Web site). http://www.mrtd.net/.

[29] Unix Manual Pages. routed - network RIP and router discovery routing daemon.

[30] David Wetherall. Active network vision and reality: lessons from a capsule-based system. In *Proc. 17th ACM Symposium on Operating Systems Principles*, pages 64–79, Kiawah Island, South Carolina, December 1999.

[31] Zebra project. GNU Zebra routing software (Web site). http://www.zebra.org/.

Using Emulation to Understand and Improve Wireless Networks and Applications

*Glenn Judd and Peter Steenkiste**
Carnegie Mellon University
Pittsburgh, PA, USA
glennj@cs.cmu.edu prs@cs.cmu.edu

Abstract

Researchers have long faced a fundamental tension between the experimental realism of wireless testbeds on one hand, and the control and repeatability of simulation on the other hand. To overcome the stark tradeoff of these traditional alternatives, we are developing a wireless emulator that enables both realistic and repeatable experimentation by leveraging physical layer emulation.

We discuss the design and implementation of a prototype wireless emulator, and show how this emulator can be leveraged to provide insight into wireless network and application behavior. Our experience shows that, compared to simulation, our emulator-based approach provides us with a better understanding of real-world wireless network performance, and enables us to quickly deploy our research into an operational wireless network, while still allowing us to enjoy the benefits of a controlled experimental environment.

1 Introduction

As wireless network deployment and use become ubiquitous, it is increasingly important to make efficient use of the finite bandwidth provided. Unfortunately, research aimed at evaluating and improving wireless network protocols and applications is hindered by the inability to perform repeatable and realistic experiments. Experimental techniques that have proven successful for wired networks are inadequate for wireless networks since a wireless physical layer fundamentally affects operation at all layers of the protocol stack in complex ways. Links are no longer constant, reliable, and physically isolated from each other, but are variable, error-prone, and share a single medium with each other and with external uncontrolled sources.

An ideal method of wireless experimentation would possess the following properties: repeatability and experimental control, layer 1-4 realism, the ability to run real applications, configurability, the ability to modify wireless device behavior, automation and remote management, support for a large number of nodes, isolation from production networks, and integration with wired networks and testbeds. We now discuss how alternative methods of experimentation fare with respect to this list of desirable properties.

The most direct method of addressing realism is to conduct experiments using real hardware and software in various real world environments. Unfortunately, this approach faces serious repeatability and control issues since the behavior of the physical layer is tightly coupled to the physical environment and precise conditions under which an experiment is conducted. The mobility of uncontrolled radio sources, physical objects, and people makes these conditions nearly impossible to reproduce. Even repeating the same experiment twice can be a daunting task when anything in the surrounding environment is in motion; remote researchers face an even bleaker situation trying to reproduce an experiment. It is also difficult to avoid affecting colocated production networks. Moreover, configurability and management of even a small number of mobile nodes distributed in three dimensions is cumbersome.

For these reasons, many researchers have understandably embraced simulation. This approach solves the problems of repeatability, configurability, manageability, modifiability, and (potentially) integration with external networks, but faces formidable obstacles in terms of realism. Wireless simulators are confronted with the difficult task of recreating the operation of a system at all layers of the network protocol stack as well as the interaction of the system in the physical environment. To make the problem tractable, simplifications are typically made throughout the implementation of the simulator. Even fundamental functions such as deciding what a received frame looks like [1] diverge greatly from the operation of real hardware. Evaluating real applications running over wireless networks is typically very difficult using a simulator. In addition, while wireless technology is undergoing rapid advances, wireless simulators, in particular open source wireless simulators, have lagged significantly behind these advances as discussed in Section 7.

The aforementioned issues with simulators, and a desire to avoid long simulation times, have caused some researchers to adopt emulation as a means of evaluation. Emulation retains simulation's advantages of repeatability and manageability, while potentially mitigating the issue of realism. Unfortunately, as discussed in Section 7, most emulators have adopted extremely simplified MAC

*This research was funded in part by the NSF under award numbers CCR-0205266 and CNS-0434824. Additional support was also provided by Intel. Glenn Judd is supported by an Intel Fellowship.

and physical layers. As the operation of these layers is fundamental to the operation of a wireless network, it is unclear that these emulators gain any realism over existing simulators.

We are developing a wireless emulator that enables both realistic and repeatable wireless experimentation by accurately emulating wireless signal propagation in a physical space. Unlike previous approaches, this emulator utilizes a real MAC layer, provides a realistic physical layer, and supports real applications while avoiding adopting an uncontrollable or locale-specific architecture. The key technique we use to accomplish this is digital emulation of signal propagation using an FPGA.

Our emulator's high degree of control and fidelity allow signal propagation to be modeled in several ways: first, widely used statistical models of signal propagation can be used; in addition, traces of observed signal propagation can be "replayed" on our emulator; lastly, manual control of signal propagation can be used to analyze behavior in artificially created situations that would be difficult or impossible to reproduce in an open system. Section 4 will discuss signal modeling in more detail.

This emulator provides an attractive middle ground between pure simulation and wireless testbeds. To a large degree, this emulator should be able to maintain the repeatability, configurability, isolation from production networks, and manageability of simulation while retaining the support for real applications and much of the realism of hardware testbeds. As a result, this emulator should provide a superior platform for wireless experimentation.

This emulator is not, however, a complete replacement for simulation and real world evaluation. Simulation is still useful in cases where a very large-scale experiment is needed or in certain cases where functionality not available in hardware is required (e.g. changing the MAC firmware). Real world evaluation is still useful when radio channel fidelity beyond the capabilities of the emulator is required, or for verifying the operation of the emulator in real-world settings.

In this paper we present the design of a physical-layer wireless emulator. We introduce the architecture of this emulator in Section 2. In Section 3 we discuss an initial proof-of-concept prototype, and our partially complete implementation of a "Version 2" emulator based on this proof-of-concept. Section 4 discusses how our emulator can be used to emulate various signal propagation environments. Using both the prototype and the Version 2 emulator, we present several experiments in Section 5 and a case study in Section 6 that demonstrate the power of our approach. Section 7 discusses related work, and Section 8 concludes our discussion.

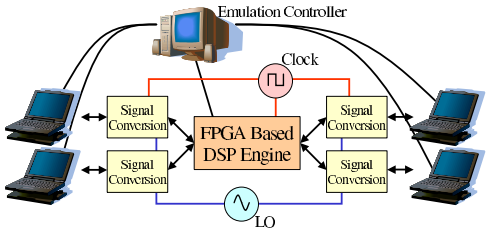


Figure 1. Emulator Architecture

2 Emulator Architecture

The architecture of our emulator is shown in Figure 1. A number of "RF nodes" (e.g. laptops, access points, cordless phones, or *any* wireless device in the supported frequency range) are connected to the emulator through a cable attached to the antenna port of their wireless line cards (each RF node corresponds to a single antenna, so a single device can be represented by multiple RF nodes). For each RF node, the RF signal transmitted by its line card is "mixed" with the local oscillator (LO) signal. This shifts the signal down to a lower frequency where it is then digitized, and fed into a DSP Engine that is built around one or more FPGAs. The DSP Engine models the effects of signal propagation (e.g. large-scale attenuation and small-scale fading) on each signal path between each RF node. Finally, for each RF node, the DSP combines the appropriately processed input signals from all the other RF nodes. This signal is then sent out to the wireless line card through the antenna port. Given the current state of technology, a DSP Engine based on a single FPGA might support over 20 wideband RF nodes. Using multiple FPGAs or lower bandwidth RF nodes, even larger systems can be built.

The operation of the emulator is managed by the Emulation Controller, which coordinates the movement of RF nodes (and possibly physical objects) in the emulated physical space. The Emulation Controller uses location information (and other factors as dictated by the signal propagation model in use) to control the emulation of signal propagation within this emulated environment. In addition, the Emulation Controller coordinates node (and object) movement in physical space with the operation of RF node applications and sending of data. Each RF node runs a small daemon that allows the Emulation Controller to control its operation via a wired network. RF nodes that are not capable of running code may require a proxy to run the daemon on their behalf.

Connecting the Emulation Controller to an external network allows remote management of the emulator. In addition, individual nodes in the emulator may be connected to external networks in order to allow emulator nodes access to the Internet at large or to allow the emulator to be used in conjunction with testbeds such as PlanetLab [2] or Emulab [3].

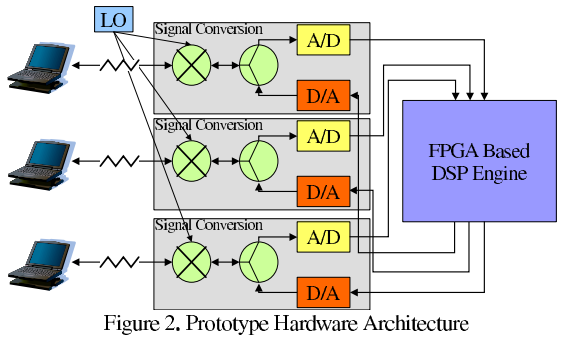


Figure 2. Prototype Hardware Architecture

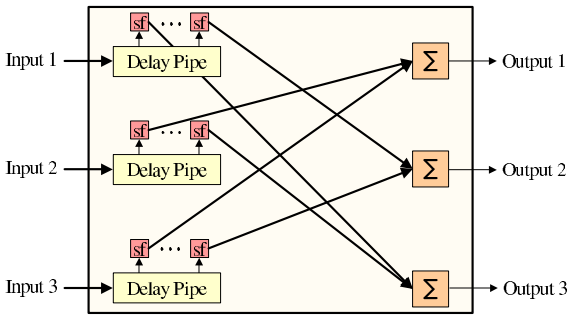


Figure 3. Prototype DSP Engine Operation

3 Implementation

To demonstrate the feasibility of the wireless emulator, we constructed a small prototype designed to validate the emulator's primary functionality by emulating signal propagation between three laptops on a single 802.11b "non-overlapping channel". The results obtained with our prototype [4], in conjunction with MIT's Roofnet project [5], and experiments discussed in Sections 5 and 6 show that the approach we advocate is capable of providing powerful wireless emulation capabilities.

We first discuss our prototype's hardware and software implementation, and then discuss how Version 2 improves on the capabilities demonstrated by the prototype.

3.1 Proof-of-Concept Prototype

Hardware. Figure 2 shows the hardware architecture of the prototype. Each laptop operates on a single 802.11b channel centered at frequency F which contains its main spectral elements from $F - 11MHz$ to $F + 11MHz$. The outgoing signal from each laptop is first attenuated and then converted to a low frequency by "mixing" each signal with an "LO" signal centered at $F - 13MHz$. The resulting output from the mixers (ignoring the signal image) is a signal ranging from 2 to 24 MHz. This signal is then fed into an A/D board for sampling. Each A/D board generates 12-bit digital samples of the incoming signal at 52 Msps, and sends them to the FPGA for processing. The output signals from the FPGA are converted to analog by the D/A and then "mixed up" and attenuated before arriving at the destination wireless card's antenna port. We used two types of wireless NICs in our prototype: antenna-less Orinoco Gold cards, and Engenius NL-2511CD Plus Ext2 Prism 2.5 based cards which both allow the connection of an external antenna or coaxial cable.

DSP Engine. As shown in Figure 3, inside the FPGA, the signals are first sent into a delay pipe where one or more copies ("taps") of the signal are pulled off after going through a programmable amount of delay. Each of these signals is then scaled by a programmable factor. Each outgoing signal, from the FPGA to an RF node, is then computed by summing the scaled signals from the other RF nodes. These outgoing signals are then sent to the D/A board for reconstruction.

The programmable nature of this circuit allows us to trade off resources such as the precise depth of the delay pipes and number of signal copies supported. Thus, we can customize the operation of the FPGA to the particular test being run.

For each signal path inside of the FPGA, the Emulation Controller discussed below is capable of dynamically adjusting both the attenuation and delay from the source to the destination by dynamically setting the scaling factors and delay mentioned previously at a rate of approximately 1,000 scale factors or 2,000 delay settings per second. Hence, for each signal path, the emulator can recreate effects such as "large-scale path loss" (a fixed attenuation that does not change unless RF node movement is emulated) and "fading" (rapid variation in signal strength that can occur even if the device antennas are motionless).

As the DSP Engine is implemented in an FPGA, the operation described above, and used in the experiments presented in this paper, may be changed as needed for particular signal propagation models. For instance, fading could be computed on the FPGA to allow for emulation of even faster fading.

Emulation Controller. The Emulation Controller controls and coordinates the operation of the DSP unit and the RF nodes, and runs in one of two modes: script or manual control.

In script mode, the Emulation Controller is driven by scripts that specify each node's movement, communication, and application behavior. As the RF nodes move about in the emulated physical space, the Emulation Controller continuously computes attenuation of each signal path and the scaling factors required to emulate this attenuation (our prototype currently uses a simple large-scale path loss model based on measurements in our local environment). After computation, these scaling factors are sent to the DSP Engine. Emulation Controller scripts can also generate network traffic between any pair of nodes, and synchronize this traffic with node movement and application behavior.

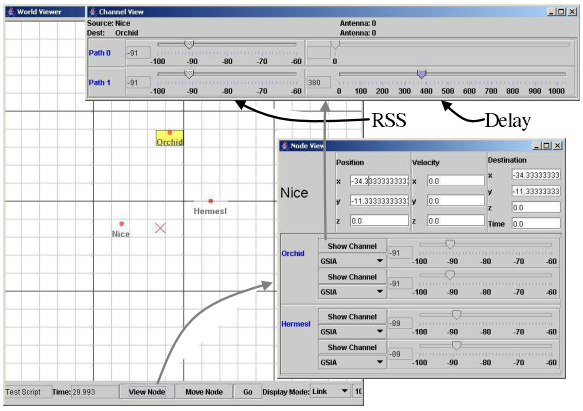


Figure 4. Emulation Controller

The Emulation Controller also generates a visual display of node location in the emulated physical environment as shown in Figure 4.

In interactive mode, the GUI shown in Figure 4 may be used to move nodes in the emulated physical environment. As shown in the "Node View" and "Channel View" windows of Figure 4, interactive mode also allows manual control of both received signal strength and delay for each channel.

The experiments we discuss in later sections make use of both the scripted and the manual control modes of the Emulation Controller.

3.2 Version 2

Our prototype emulator confirmed the power of our approach [4], and proved itself to be an extremely useful tool in its own right. Nevertheless, the scale, fidelity, and bandwidth of our prototype were limited by the fact that we used an inexpensive off-the-shelf evaluation board for the DSP Engine. The dynamic range of our emulator was limited by the prototype Signal Conversion Module's use of simple connectorized components. "Version 2" of our emulator addresses these key limitations of the prototype. We now describe this implementation; Section 3.3 then presents the results of experiments that show the fidelity of Version 2.

Our Version 2 DSP Engine is currently under development. It will have the same fundamental architecture as the prototype DSP Engine, but it will greatly improve on the prototype by using a much larger FPGA on a custom board with high-speed connectors to the Signal Conversion Modules. It will be able to support 15 RF nodes and 100 MHz of bandwidth versus 3 nodes and 25 MHz for the prototype, and will also allow for much finer grained control of signal fading.

The Version 2 Signal Conversion Module is complete and functional. A fully assembled Signal Conversion Module is shown in Figure 5. The RF Front End board on this module replaces the connectorized components used in the prototype, and increases the dynamic range

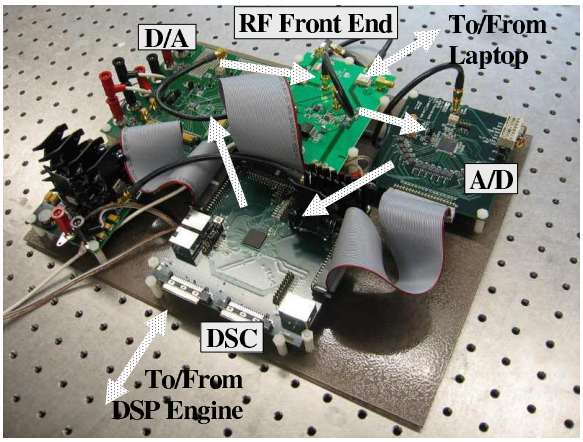


Figure 5. Production Emulator Implementation

of Version 2 to 60 dB versus 40 dB for the prototype. (Version 2 achieves 50 dB isolation from the strongest spurious signal caused during emulation). The A/D and D/A boards used in this module are capable of running at 210 Msps which is over 3 times that of the prototype. This allows us to capture around 100 MHz of bandwidth directly, and is sufficient to capture all North American 802.11b/g channels or a portion of 802.11a.

Unlike the prototype, the Version 2 Signal Conversion Module utilizes a "Digital Signal Conversion" (DSC) board. The inclusion of this board arose from the need to convert high-speed digital signals from the different signaling requirements used by the A/D, D/A, and the DSP Engine. For flexibility, this board was implemented using a modest FPGA, which allows each DSC to assist the DSP Engine in certain cases.

3.3 Validation

Experiments demonstrating the performance of our prototype were presented in [4]. We now present experiments validating the fidelity and isolation of Version 2 which show significant improvement over the prototype's performance.

As the DSP Engine operates entirely on digital signals, the fidelity of the emulator is determined by the Signal Conversion Module. Hence, we may measure the fidelity of our production emulator solely by measuring the fidelity of the Signal Conversion Module. We employed this approach by using two Signal Conversion Modules to emulate two RF signal paths. The FPGAs on the DSC boards implement the signal attenuation required for these tests. These tests used Engenius NL-2511CD Plus Ext2 wireless cards.

Fidelity. A signal's physical layer fidelity is measured by comparing it with an ideal signal; the signal is measured by periodically sampling the signal and plotting the results on a polar graph as shown in Figure 6. This is known as the signal's "constellation". (In the figure, each constellation contains four clusters of points.)

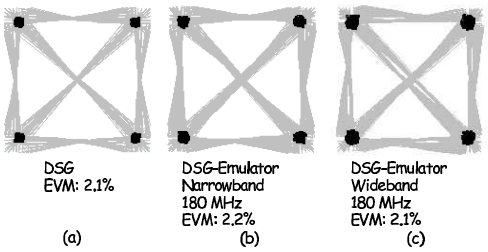


Figure 6. Physical Layer Fidelity

We can then visually compare the measured constellation against an ideal constellation.

We can quantify the difference between a measured signal and an ideal signal by measuring "error vector magnitude" (EVM). EVM is the relative difference between ideal signal constellation points and observed constellation points. EVM measures the average magnitude of the error vector (a vector from the ideal constellation point to the observed point) as a percentage of the ideal signal vector's magnitude.

Figure 6 compares the modulation fidelity of a signal generated by a digital signal generator (a) with that of the same signal passed through our production emulator (b) and (c). Comparing (a) with (b) we see that when the emulator is digitizing in narrowband mode (a single 802.11b channel) the constallation loses some crispness, but is still excellent; EVM increases slightly. (c) shows that when digitizing a wideband signal (802.11b channels 1-11) the signal degrades slightly more, but is still quite good. The EVM measurement in this case should not be regarded as saying that there is no signal degradation in wideband mode, but merely shows that the degradation is within the margin of measurement error.

Our earlier prototype work [4] demonstrated that our emulator does not distort on-card measurements such as received signal strength (RSSI). This previous work also showed that the prototype link delivery performance was close to that of a coaxial-based comparison, and that signal modeling was repeatable across experiments. We omit similar tests from this work in the interest of space.

Figure 7 demonstrates that our prototype's physical and link layer fidelity translates into transport level fidelity by comparing the TCP throughput for two laptops connected via coaxial cable and discrete attenuators versus two laptops connected via our production emulator. Each data point is an average of 20 trials measuring one-way TCP throughput for approximately 5 seconds. Confidence intervals are omitted since they are tight, and the SNR measurement error is dominant (about 1 dB). The results match quite closely and are within the measurement error of the experiment.

Isolation. An important benefit of our prototype is the ability to conduct experiments in isolation from external sources of interference. To measure this, we used a high

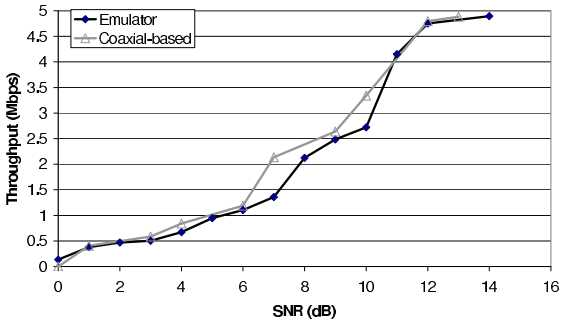


Figure 7. Transport Layer Fidelity

power source (20 dBm) external to our emulator with a strong omnidirectional antenna (5.5 dBi) to send traffic at 1Mbps. We then moved this traffic source around our immediate environment to see when our emulator could not sense any of this traffic. Our results showed that our emulator was isolated against this strong source when it was at least 10 meters away. The current limitation on this isolation is the need to sacrifice perfect shielding in order to allow the RF nodes to be cooled. Additional work should cut the interfering range down to a few meters even for strong transmitters.

Building a large setup requires that we place RF nodes in close proximity to each other. To allow for this while maintaining internal isolation, each emulator node is mounted inside of a shielded rack-mount chassis. By altering the external isolation test to measure internal isolation, we verified that nodes attached to the emulator are effectively isolated against undesired transmission to each other despite their close proximity (8.75 inches).

We next discuss how our emulator's ability to faithfully control the wireless signal is used to model signal propagation. We will then discuss several experiments that demonstrate the range of experiments enabled by our emulator.

4 Signal Propagation Modeling

With our ability to completely control wireless signal propagation comes the challenge of modeling or recreating propagation in an appropriate manner for a given experiment. Our goal in this work is not to develop and justify new physical models of signal propagation, but to discuss how current and future models as well as signal propagation trace playback can be used in our emulator.

Fortunately, unlike wireless simulators, we are freed from the task of emulating radio behavior in conjunction with signal propagation modeling: we simply pick a suitable signal propagation model, compute each receiver's received signal, and let the radio decide what happens. We do not need to make any assumptions regarding any radio issues such as "sensing range"or "interfering range".

We now discuss several different methods of modeling wireless signal propagation in our emulator. We begin with signal propagation models that require no site specific information, and then discuss models that use increasing amounts of site specific information. Some of these techniques are completely operational in our emulator (large-scale path loss, signal capture and replay), some are partially operational (small-scale fading), while others require some external tools before they can be used in our emulator (ray-tracing, channel sounding).

4.1 Large-scale Path Loss

The signal propagation model most commonly used by simulators is a large-scale path loss model. Specifically, the received signal strength at each receiver (RSS) is computed as $RSS = Pt + Gt - PL + Gr$. Where Pt and Gt are the transmit power and antenna gain at the transmitter, PL is the path loss, and Gr is the antenna gain at the receiver. Large-scale path loss models simply compute PL as a function of distance between the transmitter and the receiver.

The Emulation Controller implements large-scale path loss by simply calculating the loss between nodes whenever the distance between them changes. These loss values are then sent into the emulator where they are used to control the attenuation of the signal path between two nodes.

4.2 Small-scale Fading

While large-scale fading models can accurately capture the average path loss between two points, on a short time scale the path loss between these points may vary substantially. To support this behavior, we are currently adding the ability in our emulator to emulate this small-scale fading.

We are leveraging the technique presented in [6] to incorporate the Ricean and Raleigh statistical models of small-scale fading in our emulator. In our implementation, the fading parameters are computed offline, and are then loaded into our emulator's FPGA before emulation begins. At run time, these parameters are added to the large-scale path loss which causes short term variation with the desired statistical properties. Independent use of fading parameters should allow independent, on-line modification of small-scale fading for each RF node.

4.3 Ray Tracing

The previous two methods required no site specific information other than picking the correct path loss models and model parameters. By incorporating site-specific information, it is possible to generate more accurate signal propagation models.

One technique that can be implemented in the emulator is to leverage ray tracing techniques. If the motion of nodes can be pre-computed off-line, ray-tracing techniques can be used to precisely compute all rays incident on each receiver at a given point in time. If motion cannot be pre-computed, then approximations can be made.

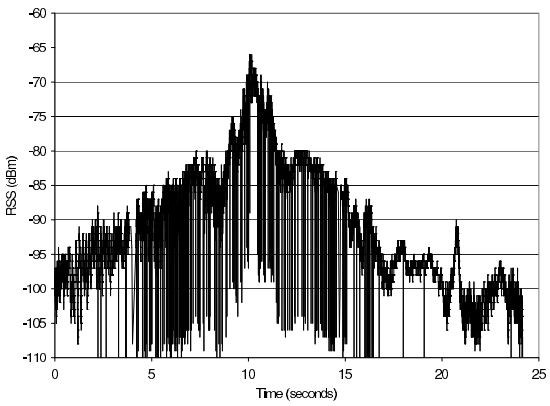


Figure 8. Freeway Driveby Measured RSS

At runtime, the pre-computed series of attenuation over time values for each signal path would then be used to set path attenuation inside the DSP Engine.

4.4 Capturing and Replaying Signal Behavior

One simple method of accurately modeling signal propagation is to measure the signal propagation in a given environment and then to replay it. We have implemented a signal capture system using standard wireless NICs that measures path loss in a physical environment. This system works by constantly sending small packets from each transmitter to be emulated and receiving these packets on each receiver being emulated.

Our emulator then simply replays the observed traces of signal strength. To demonstrate this capability, we captured path loss from a car driving along a freeway at 60 MPH to a base station located at a fixed point near the freeway.

The traffic source was a 23 dBm 802.11b source attached to a 5 dBi isotropic antenna placed on the roof of the passing car. The receiver used the same hardware as the sender, but with the antenna placed on the roof of a stationary car at the side of the freeway. As the sender passed, it continuously broadcast small 1 Mbps broadcast packets which were recorded by the receiver. The result of this test was signal strength measurements with 1 ms granularity. We then post-processed this trace to extract the timestamped signal and noise measurements.

Figure 8 shows the trace extracted using this method. Our emulator then simply reads, and recreates the observed path loss at the given time. Our current trace replaying software is limited to 2.5 ms granularity.

4.5 Channel Sounding

A more sophisticated method of measuring signal propagation in a physical environment is to use spe-

cialized hardware to precisely measure the "impulse response" of the channel. Such measurements can be difficult to obtain since they require specialized hardware. Once obtained, however, our emulator is capable of replaying these measurements by setting the attenuation and delay of each signal path in the DSP Engine according to the values extracted from the channel sounding.

4.6 Discussion

Before presenting experimental results, we briefly discuss the capabilities and limitations of signal propagation modeling using our approach.

Simulation. Many of the signal propagation models that we utilize can be also be used in simulation. This superficial similarity, however, belies a massive difference in how these models are used. Computational constraints placed on a simulator, force the simulator to work at a very coarse timescale. Our emulator, on the other hand, uses a statistical propagation model to manipulate a real modulated signal on the timescale of 5 ns. This is then sent to a real receiver to determine the reception behavior. Accurate receiver behavior in a simulator would require transistor level simulation which is completely infeasible for the number of nodes that we are looking at. Realtime simulation of such behavior is out of the question.

Similarly, while a simulator can replay a captured channel trace, it can only do so at a very coarse timescale and with far less fidelity than a physical layer emulator.

Real-world experimentation. The ability to precisely recreate a signal propagation environment is a huge advantage compared to real-world experimentation. This power, however, comes with a price of reduced realism and scale in signal propagation.

Our approach necessarily models a wireless channel using discrete elements (e.g. one line-of-sight ray and two reflections) whereas a true wireless channel is a continuous phenomenon. Also, as the number of RF nodes attached to our DSP Engine increases, the number and length of delayed signal paths that we can implement drops. Hence our approach is a compromise between the fidelity of the real-world and the control of simulation.

Noise. The term noise is frequently used to refer to both true noise (e.g. receiver noise) and interference from other wireless devices. Receiver noise is naturally present in our system since we use real receivers. Interference from other wireless devices can be supported in several ways. First, if RF Node ports are free and the devices are available, these devices can simply be attached to our emulator. Secondly, it is possible to record noise resulting from interference and to replay this in the emulator. Third, a white noise generator can be implemented in either the DSP Engine or the DSC card to generate noise.

Note that our effective receiver noise floor will be slightly higher than a coaxial based system since we use additional amplifiers etc. that introduce noise. This level will still be much lower than the noise floor of a true free-space wireless system.

Scale. As hardware is finite, the richness of channel modeling possible using hardware-based emulation drops as the scale of the network being emulated increases. The limiting factor is typically the number of multipliers in the DSP Engine's FPGA.

For much of our discussion, we have assumed the desire to support the independent pairwise emulation of all pairs of RF Nodes attached to an emulator. Clearly this approach becomes infeasible at a certain point as the complexity of pairwise interaction is order n^2.

It is important to observe, however that emulating complete interaction is not always necessary. Clearly, if nodes are out of range with respect to each other, then no emulation between them is necessary. In addition, complexity may be reduced by simplifying and aggregating the emulation of channels for distant nodes.

Multi-element Air Interface Support. Current wireless networks are pushing the limits of the throughput that are possible with a single element antenna. Future networks will increase throughput by using multiple elements to support techniques such as steerable antennas, MIMO, and "time reversal".

Our emulator can support such emerging technologies in two ways. First, where hardware exists, our emulator can support these multi-element experiments by simply treating each element as an independent RF node. The control software then simply controls these RF nodes in a coordinated fashion which also opens up some room for reducing FPGA resources consumed. Second, in certain circumstances, it may be possible for the emulator to emulate the effect of a given technology. For instance, a steerable antenna can be completely emulated without necessarily using a true steerable NIC.

5 Experiments

Our emulator enables a broad set of experiments to be conducted in a controlled and automated environment. To give a feel for the power of our emulator as a research tool, we now present several experiments that illustrate various types experimentation that our emulator enables.

We first discuss how our emulator can improve understanding of the impact of the physical layer on higher layers. We then discuss our emulator's support for emerging antenna and air interface technologies. Finally, we discuss how our emulator can be used to conduct micro and system level benchmarks of wireless performance. Section 6 will then present a case study showing how our

emulator can be used to analyze a wireless protocol improvement.

These experiments were all conducted using one or more of three RF Nodes connected to our prototype: "Orchid", "Hermes", and an interferer ("Nice" or a Bluetooth source). For experiments conducted in an emulated physical environment (i.e. where manual control of channel parameters is not required), we use a log-based path loss model derived from our local environment. For each of the experiments discussed, obtaining realistic results using traditional methods would be difficult or inaccurate.

5.1 Physical Layer Impact on Higher Layer Performance

5.1.1 Hidden Terminal



Figure 9. Hidden Terminal Topology

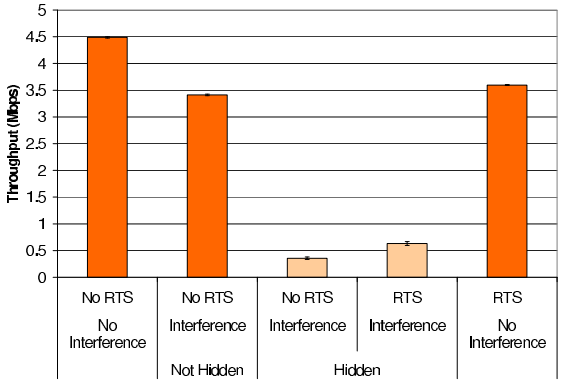


Figure 10. Hidden Terminal Results

A well known example of a low layer issue that has potentially serious ramifications for application performance in wireless networks is the "hidden terminal" problem. Evaluating the hidden terminal problem in a real world environment is troublesome since it is difficult to determine if nodes are in carrier sensing range of each other. Moreover, carrier sensing range constantly fluctuates in the real world. This experiment highlights our prototype's ability to overcome these difficulties by providing precise, independent control over the signal paths between all nodes. This allows us to evaluate the hidden terminal problem by simply commanding the emulator to "disconnect" the desired nodes while leaving the communication between other nodes unaffected.

As illustrated in Figure 9 we arranged our three nodes in a line with all nodes in range of each other. (For simplicity we will speak of spatial relationships in our virtual physical environment as if they were based in a real physical environment). We then measured TCP throughput from Hermes to Orchid while Nice was used to generate interfering traffic using a unicast ping flood directed at Orchid. Orinoco cards were used for these tests.

As shown in the Figure 10 "No RTS, No Interference" test, throughput between Orchid and Hermes is excellent when there is no interference (each value is an average of 25 trials with 95% confidence intervals shown). In the "No RTS, Interference, Not Hidden" test, we see that when Nice begins interfering, throughput is still quite good (ping packets are much smaller than the TCP packets).

We then created a hidden terminal situation by artificially "severing" the link between Hermes and Nice while leaving the other communication paths unaffected. (The ability to create a hidden terminal situation without "moving" the nodes allows us to directly compare results between the hidden and non-hidden tests.) The "No RTS, Interference, Hidden" test shows that throughput between Orchid and Hermes drops dramatically in this case.

We next analyzed the efficacy of 802.11's RTS/CTS mechanism at overcoming the hidden terminal problem by repeating the previous tests with Hermes set to always use RTS/CTS for frames over 200 bytes. The "RTS, Interference, Hidden" test shows that RTS/CTS is able to double throughput; nevertheless throughput is still much lower than when the interferer was not hidden. Comparing the final "RTS, No Interference" test with the "No RTS, No Interference" case shows that the overhead of RTS/CTS alone is roughly 1 Mbps. Further investigation (and coaxial-based verification) revealed that the cause of this underwhelming improvement was the failure of RTS/CTS to prevent rate fallback. The ability to analyze this type of subtle behavior in a controlled environment is a key advantage of our emulator.

5.1.2 External Interference

Another well known problem that can afflict wireless networks in a license free band is interference from external sources. To illustrate our ability to investigate interference from arbitrary sources we conducted a simple experiment involving two 802.11b nodes communicating in the face of interference from a Bluetooth source. As shown in Figure 11, each node was positioned 50 meters from the other two nodes.

Figure 12 shows the results of communication between Hermes and Orchid for four scenarios (each value is an average of 25 trials with 95% confidence intervals shown), two of which - the "Yagi" cases - will be discussed in the next section.

In the "Isotropic, No Interference" test, Hermes and Orchid communicate with omnidirectional antennas with no interference (using a TCP benchmark with traffic from Orchid to Hermes). Communication is only around 1.25 Mbps due to the distance between the nodes.

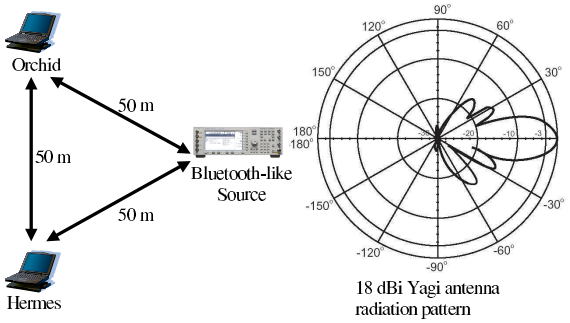


Figure 11. Directional Antenna Topology

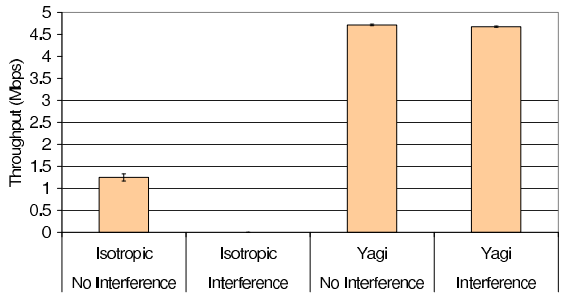


Figure 12. Directional Antenna Results

In the "Isotropic, Interference" test, Hermes and Orchid communicate as before, but the Bluetooth source is configured to broadcast a constant 15 dBm signal with Bluetooth modulation. TCP communication between Orchid and Hermes is not possible in this case.

5.2 Flexible Antenna and Multi-element Air Interface Support

Complete control over signal propagation also allows our prototype to emulate arbitrary types of antennas. To illustrate this, we analyzed the ability of directional antennas to improve range and spatial reuse by minimizing the effects of interfering Bluetooth traffic (discussed in 5.1.2). Orinoco cards were used for these tests.

The "Yagi" tests repeat the "Isotropic" tests discussed previously, but with 18 dBi Yagi antennas [7] attached to Orchid and Hermes. These antennas are aimed directly at each other. Figure 11 shows the radiation pattern for these antennas. Note that for Orchid and Hermes, the Bluetooth source lies along a side lobe with approximately 22 dB and 18 dB respectively less gain than the primary lobe. As shown in Figure 12 these directional antennas successfully increase the communication rate and also mitigate the effects of external interference.

5.3 Benchmark Experiments

We now consider "benchmark" experiments that are designed to measure particular aspects of wireless NIC or link behavior. In additon to providing the control necessary for these tests, the emulator allows these tests to be automated which greatly reduces execution time while eliminating the error associated with manually conducting similar experiments.

These capabilities also enabled us to compare wireless link behavior observed in Roofnet [5] against link behavior in a controlled emulated environment.[1]

5.3.1 NIC Signal Measurement Characterization

Many researchers have proposed techniques that rely on signal strength and/or noise floor measurements provided by the card. Two common examples are signal strength based device location [8] and SNR based rate selection [9]. The success of these proposed techniques hinges on the accuracy of NIC signal measurement; very little information, however, has been published regarding the accuracy of these measurements in actual hardware.

To investigate the accuracy of signal measurements made by current 802.11b cards, we tested the measurement behavior of five wireless cards. Each card was the exact same model: an Engenius NL-2511CD Plus Ext2 card. Using our emulator to connect a single transmitter-receiver pair we were able to precisely control the received signal strength (RSS) at each card (we held the transmitter constant while alternately measuring each receiver). For each signal strength between -70 dBm and -100 dBm at 2 dB intervals we sent 500 packets of 1500 bytes each at 1 Mbps. We then computed the average signal strength (RSSI) and noise measured by each card (along with 95 % confidance intervals).

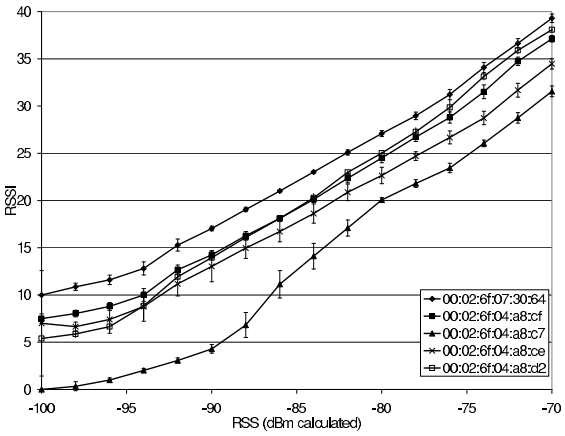


Figure 13. Per-card RSSI Variation

As shown in Figure 13 there is approximately 10 dB of variation in the measurements even for the exact same model of card. This is clearly inadequate for many purposes. For most cards, however, this variation seems to be caused by a constant bias. This implies that each card's measurement behavior, RSSI, for a given RSS can be defined as: $RSSI(RSS) = RSS + Ec + E(RSS)$. Where RSSI is the measured signal strength, RSS is the actual signal strength, Ec is a constant (per-card) error term, and E(RSS) is each card's variation of from the

[1] This was done in conjunction with the Roofnet project

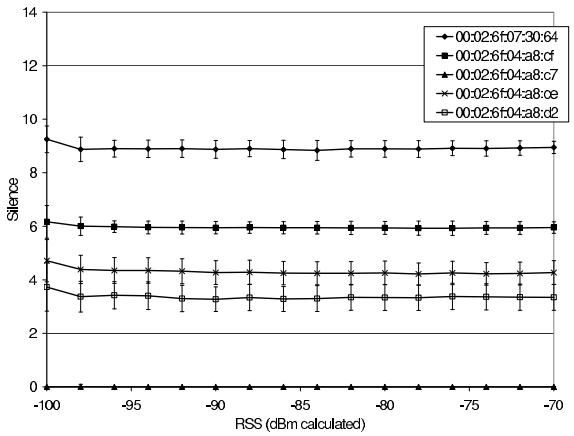


Figure 14. Per-card Noise Variation

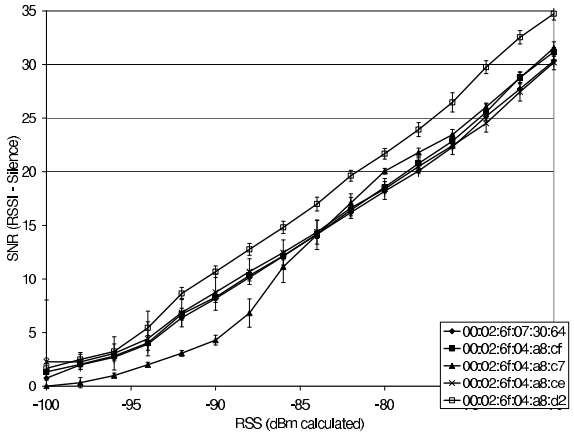


Figure 15. Per-card RSS Variation after Correction

base Ec for a particular RSS. Ideally, each card would have a lookup table that would give the Ec as well as E(RSS) for each RSS. Lacking such a table, however, we can leverage the fact that most of the error is contained in Ec to correct RSSI.

One very simple method of obtaining a good estimate of Ec is to min-filter the noise measurements (the filtering eliminates spurious noise measurements). As shown, in Figure 14, the noise measurements over the same set of tests shows very similar variation. That is, each card's variation in RSSI closely matches it's variation in measured noise. Figure 15 shows the variation in RSS when using this technique. With the exception of one card, this lowers the variation to approximately 4 dB. This is a greatly reduced variation, but may not be low enough for some purposes (e.g. signal strength based location). Complete card characterization of the relationship between RSSI and RSS is possible, but may not be worth the per-card testing required.

5.3.2 NIC Delivery Rate Variation

We next measured the 1 Mbps packet delivery rates for the same five cards discussed previously. We report delivery rate as the fraction of transmitted packets that were

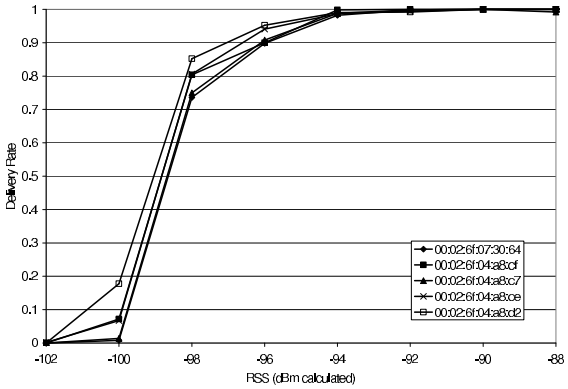


Figure 16. Per-card Delivery Rate Variation

received error free. We used the same experimental setup described in 5.3.1 with the exception that we varied RSS between -70 dBm and -102 dBm (we omit tests above -88 dBm as there was no loss).

As shown in Figure 16, there seems to be less variation in delivery rates than in RSSI. Significantly, the delivery rate performance measured roughly follows the noise measurements in Figure 14: cards reporting lower noise levels tend to have a higher delivery rate. Hence, some of the noise floor measurement variation appears to be due to real variation in the noise floors of the NICs. This is probably due to variation in the amount of noise generated by each NIC's low noise amplifier.

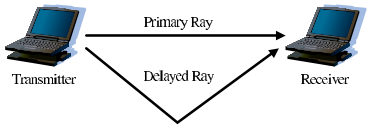


Figure 17. Two-ray Test Topology

5.3.3 Multipath Performance

We now examine card performance in the presence multipath. To do this we configured our emulator to emulate the signal propagation environment shown Figure 17 using three different primary ray strengths (-70 dBm, -90 dBm, and -95 dBm). For each primary ray strength, we caused a delayed ray to be emulated at all 2 dB increments of attenuation between the primary ray strength and -100 dBm. For each primary ray, secondary ray signal strength combination, we varied the secondary ray's delay between 0 and 2.22 us in 0.0185 us increments. For each of these combinations, we conducted a test by transmitting for 500 packets, of 1500-bytes each, from the sender. The receiver then measured the packet delivery rate and other on-card statistics such as signal and noise measurements (for successfully received frames).

RSSI measurements from this test (omitted in the interest of space) showed that RSSI measured the sum of all signals incident to the receiver, and was fairly insensitive to the delay between the signals. The only significant exceptions being when the delayed ray completely cancelled out the primary ray.

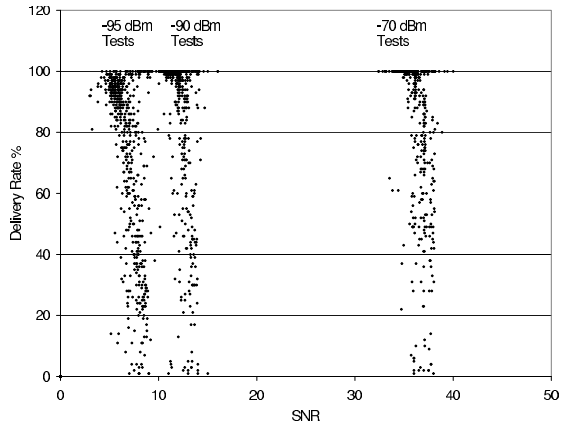


Figure 18. Two-ray Delivery Rate vs. SNR

As seen in Figure 18, the delivery rate exhibited large variation for different delay spread, delayed signal strength combinations (each point represents a the delivery rate for one primary ray strength, delayed ray strength, delay spread combination). Hence, SNR may be a very poor indicator of packet delivery rate when significant multipath is present.

We next analyzed the potential of applications to estimate the amount of multipath present using information obtained from the NIC's equalizer. On the Engenius NL-2511CD Plus Ext2 cards (and all other cards based on the same chipset), a register - "MPMetric" - is available to estimate the amount of multipath interference present during reception.

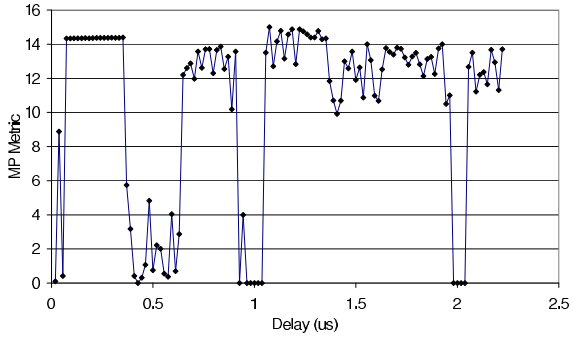


Figure 19. Two-ray MP Metric vs. Delay

As the documentation on the Prism 2.5 MPMetric register is scant, our emulator's ability to measure the behavior of this register is critical in understanding its performance. Figure 19 shows MPMetric as a function of delay spread for two equal-strength rays. These measurements were obtained from the two-ray test described earlier, and use the five Engenius cards used previously. From this test, we infer that if significant multipath reception is present, MPMetric is likely to be high. We then measured MPMetric in the presence of no multipath as shown in Figure 20. From this test we see that the MPMetric register may also go high whenever the

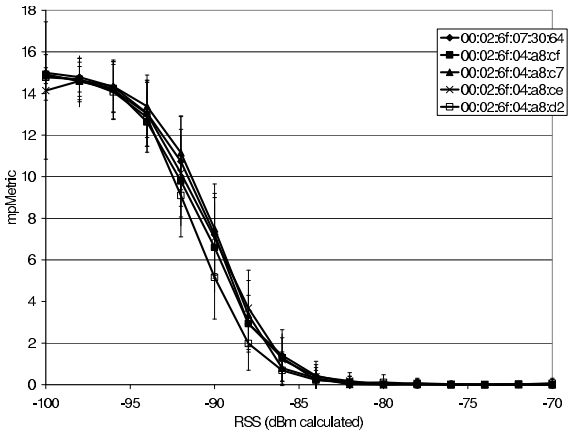


Figure 20. One-ray MP Metric vs. RSS

signal conditions are marginal irrespective of multipath. This suggests that a high MPMetric reading is a likely indicator of multipath when the received signal strength is high, but it is not a useful indicator of multipath when the received signal strength is weak.

6 Case Study: 802.11b Rate Selection

We now present a small case study that demonstrates how our emulator can be used to analyze and improve wireless protocol performance.

When selecting a transmit rate, a fundamental tradeoff that wireless protocols must make is throughput vs. range: higher transmit rates increase throughput but at the cost of range and robustness to interference. Rather than selecting a fixed point in this tradeoff, wireless protocols such as 802.11b support multiple transmit rates. This allows wireless NICs to potentially select the best transmit rate in a given environment and at a given moment.

Selecting the best rate, however, is a difficult problem and several schemes have been proposed. Our emulator allows a controlled comparison of the performance of these schemes on real hardware. For illustrative purposes we examine three schemes: ARF - auto rate fallback, SNR signal-to-noise ratio based scheme, and ERF - Estimated Rate Fallback. We describe each of these approaches below.

We based our transmission rate selection implementations on the HostAP mode Prism driver for Linux. We made extensive alterations in order to take fine-grained control of rate selection out of the firmware, and put it into the driver. These alterations give us per-packet control over transmit rate, and effectively disable firmware rate control.

ARF Implementation. Auto rate fallback attempts to select the best transmit rate via in-band probing using 802.11's ACK mechanism. ARF assumes that a failed transmission indicates a transmit rate that is too high. A successful transmission is assumed to indicate the cur-

rent transmit rate is good, and that a higher rate might possibly be useful.

Our ARF implementation works as follows. If a given number of consecutive packets are sent, then increment to the next highest transmission rate. If a given consecutive number of packets are dropped then decrement the rate. If no traffic has been sent for a given amount of time, then use the highest possible transmission rate for the next transmission. In our implementation, the increment threshold is set at 6, the decrement threshold at 3, and the timeout value at 10 seconds. (The Prism 2.5 firmware based ARF algorithm uses a decrement threshold of 3 and a timeout of 10 seconds, but is somewhat different than our algorithm since retries are implemented entirely in firmware.)

SNR Implementation. SNR based approaches attempt to eliminate the overhead of probing for the correct transmission rate by selecting the optimal transmission rate for a given SNR. These schemes typically ignore multipath interference, and assume that card RSSI/noise floor measurements are completely characterized on a per-card basis.

SNR based rate selection algorithms are faced with the fundamental problem that the information they need to make the rate selection decision is measured at the receiver. Our SNR based implementation leverages receiver based reception information, like RBAR [9], but eliminates the per-packet overhead and works with standard 802.11. The key insight that our SNR based algorithm leverages is the fact that instantaneous path loss between two given points is symmetric in both the sending and receiving directions [2]. Hence, it's possible to estimate SNR at the receiver by observing traffic in the reverse direction. We omit further details of this scheme as they are beyond the scope of this paper.

Estimated Rate Fallback. While signal based transmission rate selection has the benefit of quickly setting the transmission rate, this technique may be inadequate in some situations. Auto rate fallback, on the other hand, has the advantage of implicitly taking all relevant channel factors into consideration, but may probe more than necessary. We developed a simple hybrid algorithm that uses both SNR and ARF in conjunction the on-card measurements of multipath. We call our scheme Estimated Rate Fallback (ERF).

The basic idea of ERF is to run the ARF and SNR based schemes in parallel, and then to select the appropriate estimate. We do this by using the SNR based estimate unless one of the following is true: multipath is detected, or the SNR estimate is near a decision threshold (2 dB in our implementation). This allows ERF avoid the multipath weakness of the SNR based approach while reducing the need for card characterization.

[2]We assume a single receive and transmit antenna. Our approach can be modified to support the general case.

Rate Selection Algorithm Comparison We now evaluate the performance of the previously discussed transmission rate selection algorithms using three emulated signal propagation environments. In all cases, we use the same test to measure performance.

Under lightly loaded traffic conditions, optimal rate selection is not strictly necessary since a lower transmission rate can simply be used. Rate selection becomes critically important, however, when the wireless network is running at capacity. For two of our tests, we examine this fully loaded condition for a single transmit-receive pair. For the third test, we examine a lightly loaded situation.

To measure performance of a single transmitter under full load, we transmitted as many unicast UDP 1400-byte packets as possible from the transmitting node to the receiving node under the given signal environment. For the lightly loaded scenario, we sent 100 packets over 10 seconds and measured the number successfully received.

These tests highlight the emulator's ability to enable controlled comparison of rate selection mechanisms with a high degree of repeatability. For each experiment we briefly discuss how the experiment would have fared using an alternate approach.

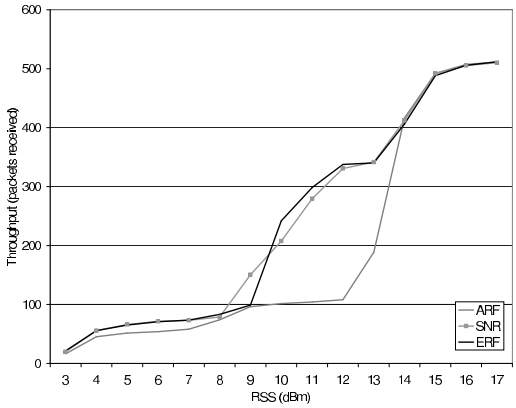


Figure 21. Rate Selection for Fixed RSS

Fixed RSS. The first test that we conducted to evaluate our rate selection mechanisms was to measure performance when the received strength was constant and the source sent as much traffic as possible as described above. Figure 21 shows our results. As expected, SNR performs well. ARF, on the other hand, performs poorly at intermediate signal levels where it is periodically probing for a higher bandwidth that will never be useful. ERF, is able to match SNR performance quite closely.

Obtaining this result using real-world experimentation would be possible, but tedious since positioning nodes to obtain a particular fixed RSS is difficult. Simulation might be used, but would only yield useful results if the hardware were modeled accurately.

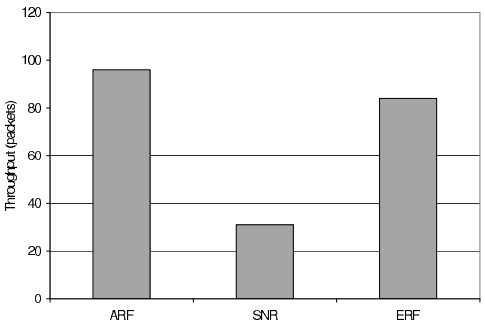


Figure 22. Rate Selection for Under Multipath

Multipath. Next, we measured rate selection performance under in a multipath environment by commanding the emulator to introduce a delayed copy of the primary signal from the sender to the receiver (ideally this would be both directions) with a fixed delay of 1 symbol period. With the RSS of the primary ray set to -77 dBm, we set the delayed ray strength to -84 dBm. As shown in Figure 22, ERF and ARF perform much better than SNR since SNR sends at 11 Mbps. This also masks the fact that SNR uses multiple retries to even attain this throughput. This test demonstrates that multipath can cause the SNR based scheme to fail, although it is unclear whether this situation is common enough to worry about in many environments. Nevertheless, ERF is able to use hardware information to eliminate even this situation.

Eliciting this result using real-world experimentation would essentially require a highly controlled large-scale RF test range. Using simulation would simply not be feasible.

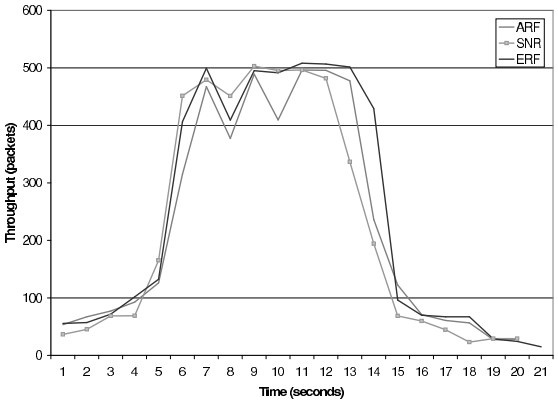


Figure 23. Rate Selection for Driveby Emulation

Fast Fading. We next tested performance in a fast fading environment, by measuring throughput during a replay of a "drive by" scenario similar to that shown in Figure 8. (In this experiment, we are simply emulating the fast fading caused by multipath, and are not actually emulating multiple signal copies. Hence, the multipath differences in the various algorithms are not demonstrated by this experiment.) Figure 23 shows that in this scenario, all algorithms perform similarly though ARF and ERF generally outperform SNR when the signal is marginal, while SNR and ERF generally outperform ARF when the signal is strong.

This experiment demonstrates the benefits of being able to replay the exact same signal trace. Comparing these rate selection algorithms in a real drive-by experiment would be difficult since even slight variations in mobility would cause channel inconsistency across experiments. Hence, it would be difficult to separate the effects on performance due to the different algorithms from the effects due to RF channel variation.

In practice, experiments that include mobility are also very cumbersome to execute in the real-world especially as the number of mobile nodes increases.

A simulated test would result in a much coarser grained use of the signal fading trace and fail to simulate the effects of rapid fading due to vehicle mobility. Hence, confidence in the accuracy of such a simulated test would be greatly reduced.

7 Related Work

7.1 Wireless Simulators

For several years now, ns-2 [10] has been the de facto standard means of experimental evaluation for the wireless networking community. Yet ns-2's wireless support has not kept pace with current technology, and is targeted towards the original 802.11 standard developed in 1997. Even this support, however, is inexact as ns-2 does not support automatic rate selection, uses a non-standard preamble, and a non-standard 802.11 ACK timeout value. In addition, ns-2's physical layer is particularly simple [1]. As a result, some researchers are opting to use commercial simulators such as QualNet [11] and OpNet [12] since they claim better support for current standards. Despite these claims, however, it is unclear how well these simulators reflect actual hardware.

7.2 Wireless Emulators

Emulation has proven to be a useful technique in wired networking research [3, 13, 14], and it has an even larger potential in the wireless domain.

A common approach that has been taken for wireless emulation [15, 16, 17] is to capture the behavior of a wireless network in terms of parameters such as capacity and error rates and then use a wired network to emulate this behavior. This has the advantage of allowing the use of real endpoints running real applications in real time. The wireless MAC and physical layers, however, are only very crudely simulated. For this reason, it is unclear whether or not this approach can obtain more realistic results than pure simulation.

RAMON [18] uses three programmable attenuators to allow emulation of the signals between a single mobile node and two base stations. While useful for the intended application of mobile IP roaming investigation, the inability to independently control all signal paths severely limits this approach.

7.3 Wireless Testbeds

More recently, several efforts such as Emulab [19], WHYNET [20], Orbit [21], and MiNT [22] have begun using controlled wireless testbeds. Though they mitigate some of the issues with respect to control and isolation, these approaches still inherit the benefits and shortcomings of testbeds discussed in Section 1. In contrast, our approach allows for much finer grained and repeatable control of the physical layer.

7.4 Channel Emulators / Fading Simulators

The most functionally similar approach to the wireless emulator that we are developing is provided by commercial channel emulators [23, 24]. The goal of these emulators, however, is quite different. Rather than supporting emulation of all channels in a wireless network, commercial channel emulators are designed to support very fine-grained emulation of the wireless channel between either a pair of devices or between a small number of base stations and a small number of mobile devices (with the total of both typically being less than 8). In addition, these emulators lack direct support for half-duplex nodes and require external components to support half-duplex nodes. As a result, while these emulators are very useful for equipment vendors evaluating a new device, the limited scale, lack of support for complete interaction between all nodes, and high cost make commercial channel emulators an unattractive option.

8 Conclusion

Understanding and improving wireless network and application performance is increasingly important. Unfortunately, repeatable experimentation with real wireless nodes running real applictions operating in a physical environment is not feasible. For this reason, most wireless research has relied on evaluation via simulation. Wireless simulators do not, however, completely duplicate real hardware in an operational environment, and the correctness of wireless simulation is difficult to validate.

We have addressed these obstacles by developing a physically accurate wireless emulator that supports real applications running on real wireless devices. We have shown that this approach allows us to achieve fine grained control over RF propagation. We have demonstrated that this enables the analysis of higher layer performance in real networks and facilitates the development and evaluation of enhanced wireless protocols.

References

[1] M. Takai and J. Martin. Effects of Wireless Physical Layer Modeling in Mobile Ad Hoc Networks. *Proc. of MobiHOC 2001. Long Beach, CA*, October 2001.

[2] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. *Proc. of HotNets-I*, October 2002.

[3] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. *Proc. of OSDI 2002*, December 2002.

[4] G. Judd and P. Steenkiste. Repeatable and Realistic Wireless Experimentation through Physical Emulation. *Proc. of HotNets-II. Cambridge, MA*, November 2003.

[5] D. Aguayo, J. Bicket, S. Biswas, G. Judd, and R. Morris. Link-level Measurements from an 802.11b Mesh Network. *Proc. of SIGCOMM 2004. Portland, OR*, August 2004.

[6] R. Punnoose, P. Nikitin, J. Broch, and D. Stancil. "optimizing wireless network protocols using real-time predictive propagation modeling". *Proc. of the Radio and Wireless Conference (RAWCON) 1999*, August 1999.

[7] SmartAnt Telecomm Ltd. Yagi antennas, `http://www.smartant.com/Products/ISM/2.4G/FYW24-01518BFL.pdf`.

[8] P. Bahl, V. N. Padmanabhan, and A. Balachandran. A software system for locating mobile users: Design, evaluation, and lessons. Tech. Rep. MSR-TR-2000-12, Microsoft Corporation, February 2000.

[9] G. Holland, N. Vaidya, and P. Bahl. A Rate-Adaptive MAC Protocol for Multi-hop Wireless Networks. *Proceedings of MobiCom2001. Rome, Italy*, September 2001.

[10] S. McCanne and S. Floyd. UCB/LBNL/VINT Network Simulator - ns (version 2), April 1999, `http://www.isi.edu/nsnam/ns/`.

[11] Scalable Network Tech. Qualnet, `http://www.scalable-networks.com/`.

[12] OPNET Tech. Opnet, `http://www.opnet.com`.

[13] K. Fall. Network emulation in the vint/ns simulator. *Proc. of The Fourth IEEE Symposium on Computers and Communications*, July 1999.

[14] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. "scalability and accuracy in a large-scale network emulator". *Proc. of OSDI 2002*, Dec. 2002.

[15] B. Noble, M. Satyanarayanan, G. Nguyen, and R. Katz. Trace-based mobile network emulation. *Proc. of SIGCOMM 1997*, September 1997.

[16] P. Mahadevan, K. Yocum, and A. Vahdat. Emulating large-scale wireless networks using modelnet. *Poster and Abstract Mobicom 2002*, September 2002.

[17] T. Lin, S. Midkiff, and J. Park. A dynamic topology switch for the emulation of wireless mobile ad hoc networks. *Proc. of the 27th Annual IEEE Conference on Local Computer Networks (LCN'02)*, November 2002.

[18] E. Hernandez and S. Helal. "ramon: Rapid mobility network emulator". *Proc. of the 27th IEEE Conference on Local Computer Networks (LCN'02)*, November 2002.

[19] B. White, J. Lepreau, and S. Guruprasad. Lowering the barrier to wireless and mobile experimentation. *Proc. of HotNets-I*, October 2002.

[20] WHYNET. Whynet, `http://chenyen.cs.ucla.edu/projects/whynet/`.

[21] D. Raychaudhuri, I. Seskar, M. Ott, S. Ganu, K. Ramachandran, H. Kremo, R. Siracusa, H. Liu, and M. Singh. Overview of the ORBIT Radio Grid Testbed for Evaluation of Next-Generation Wireless Network Protocols. *Proc. of WCNC 2005. New Orleans, LA*, March 2005.

[22] S. S. P. De, A. Raniwala and T. Chiueh. MiNT: A Miniatuirized Network Testbed for Mobile Wireless Research. *Proc. of Infocom 2005. Miami, FL*, March 2005.

[23] PROPSim. Propsim c8 wideband multichannel simulator, `http://www.propsim.net/`.

[24] Spirent Communications. Tas4500 flex5 rf channel emulator, `http://www.spirent-communications.com/`.

Geographic Routing Made Practical

Young-Jin Kim[†] Ramesh Govindan[†]
[†]*University of Southern California*
Los Angeles, CA 90089
{youngjki, ramesh}@usc.edu

Brad Karp[*]
[*]*Intel Research/CMU*
Pittsburgh, PA 15213
bkarp@cs.cmu.edu

Scott Shenker[‡]
[‡]*UCB/ICSI*
Berkeley, CA 94704
shenker@icsi.berkeley.edu

Abstract

Geographic routing has been widely hailed as the most promising approach to generally scalable wireless routing. However, the correctness of all currently proposed geographic routing algorithms relies on idealized assumptions about radios and their resulting connectivity graphs. We use testbed measurements to show that these idealized assumptions are grossly violated by real radios, and that these violations cause persistent failures in geographic routing, even on static topologies. Having identified this problem, we then fix it by proposing the Cross-Link Detection Protocol (CLDP), which enables provably correct geographic routing on *arbitrary* connectivity graphs. We confirm in simulation and further testbed measurements that CLDP is not only correct but practical: it incurs low overhead, exhibits low path stretch, always succeeds in real, static wireless networks, and converges quickly after topology changes.

1 Introduction

There is a very broad literature on geographic routing algorithms, particularly on the sub-class that uses face routing on a planar subgraph [2, 7, 13, 17, 18, 24]. These algorithms are attractive for wireless ad hoc networks because they have been shown to scale better than other alternatives: they require per-node state independent of network size, dependent only on network density. More recently, geographic routing algorithms have been proposed for use as a routing primitive for static sensor networks, as building blocks for data storage and flexible query processing in sensor networks [20, 23], and even as a fallback routing mechanism for reduced state routing in the Internet [9].

Despite research activity on geographic routing spanning half a decade, we know of no work in which researchers have *implemented* and *deployed* geographic routing protocols in realistic environments. Using our implementation of the GPSR geographic routing algorithm [13]—which we believe to be the first of its kind—we first show that GPSR incurs permanent packet delivery failures between node pairs on two different sensor network testbeds where we had no control over node placement. To wit, GPSR leaves over 30% of node pairs permanently disconnected in one testbed experiment, and over 10% disconnected in another. The significant incidence of these delivery failures and their permanent nature suggest that known geographic routing techniques are impractical for use in real deployments.

GPSR is built upon graph planarization algorithms that are amenable to distributed implementation [2, 13]. These planarization algorithms rely purely on neighbor location information to determine whether or not links to neighbors belong in the planarized subgraph. When greedy forwarding is impossible, GPSR delivers a packet by successively traversing the faces of the planar subgraph cut by the line between the packet's source and destination. A body of subsequent work (including GOAFR+ [17] and its many variants) has extended this face routing technique to offer shorter worst-case paths than GPSR. A common assumption made by the planarization algorithms used by all these geographic routing protocols is that connectivity between nodes can be described by *unit graphs.* In such graphs, a node is always connected to all nodes within its fixed, "nominal" radio range, and never connected to nodes outside this range.

We show that our implementation of GPSR incurs permanent delivery failures precisely because real radios routinely violate the unit graph assumption. Such violations can cause three kinds of pathologies in the planarization process: a link in the planar subgraph is removed when it should not be (partitioned planar subgraph); the nodes at the two ends of a link disagree on whether or not the link belongs in the planar graph (unidirectional links); or a pair of crossed links remain in the supposedly planar subgraph (crossing links). These pathologies, in turn, can result in persistent routing failures in the network, where geographic routing fails to

find a path for at least one source-destination pair. A previously proposed "fix" to these planarization techniques, the mutual-witness procedure [11, 12, 24], fails to eliminate many instances of routing failure on our testbeds.

We remedy this problem by proposing a distributed Cross-Link Detection Protocol (CLDP) that, given an arbitrary connected graph, produces a subgraph on which face traversal cannot cause a routing failure, regardless of radio irregularities and localization errors. In CLDP, each node probes the faces on which each of its links sits to determine if there exists a crossing link. Crossing links are eliminated only when doing so would not disconnect the resulting subgraph. This algorithm is *qualitatively* different from the planarization algorithms used by earlier face routing protocols, in both its approach and its correctness. The unmodified GPSR algorithm conducts perimeter-mode forwarding using the subgraph produced by CLDP.[1] CLDP retains geographic routing's desirable scaling properties. Moreover, we have proven that CLDP prevents routing failures in an arbitrary connected graph.[2]

Finally, we present measurements from simulations and experiments on two different wireless sensor network testbeds that validate CLDP's correctness, and show that CLDP incurs moderate overhead, converges quickly, and picks low-loss paths. Because CLDP renders geographic routing correct on real radio networks, we believe it represents the first generally scalable and *practical* approach for any-to-any routing in large-scale wireless settings.

2 Preliminaries and Related Work

We now review prior work in geographic routing protocols and describe the essentials of the workings of geographic routing that provide the context for our work.

There is a very broad literature on geographic routing: from initial sketches suggesting routing using position information [4, 15]; to the first detailed proposals, including GFG [2], GPSR [13], and the GOAFR+ family of algorithms [17]; to refinements of these proposals for efficiency [7], robustness under real network conditions [18, 24], and even routing geographically when node location information is unavailable [21, 22].

We now describe the shared characteristics of the GFG, GPSR, and GOAFR+ algorithms, and hereafter refer to this family of algorithms simply as geographic routing.[3]

Geographic routing schemes use *greedy routing* where possible. In greedy routing, packets are stamped with the positions of their destinations; all nodes know their own positions; and a node forwards a packet to its neighbor that is geographically closest to the destination, *so long as that neighbor is closer to the destination. Local maxima* may exist where no neighbor is closer to the destination. In such cases, greedy forwarding fails, and another strategy must be used to continue making progress toward the destination. In particular, the packet must only find its way to a node closer to the destination than the local maximum; at that point, greedy routing may once again make progress.

In the case where a network graph has no crossing edges[4]—that is, the graph is *planar*— geographic routing schemes recover similarly by *face routing*. Note that a planar graph consists of *faces,* enclosed polygonal regions bounded by edges. Geographic routing uses two primitives to traverse planar graphs: the *right-hand rule*, and *face changes*. The right-hand rule tours a face endlessly in a cycle, and can thus be used to walk a face. Figure 1 shows an example of the rule, which dictates that upon receiving a packet on a link, the receiving node forwards that packet on the first link it finds after sweeping counter-clockwise about itself from the ingress link.

Consider the planar graph in Figure 2, in which the source node S and destination node D are indicated. Observe that the line segment $\overline{SD}$ *must* cut a series of faces in the planar graph; these faces are numbered and bordered in bold. Geographic routing algorithms exploit this property by successively walking the faces cut by this line. That is, they use the right-hand rule to tour a face. While walking a face, upon encountering an edge that crosses the line segment $\overline{SD}$ at a point closer to D than the point at which the current face was entered, geographic routing algorithms perform a *face change:* they begin walking the bordering face that is next along the line segment $\overline{SD}$.[5] The numbering of faces in Figure 2 shows the order in which faces are traversed from S to D on that planar graph. Should a face be toured in its entirety without discovering an edge that crosses line segment $\overline{SD}$ at a point closer to D than the point at which the current face was entered, face routing fails. On a planar graph, such a loop on a face only occurs when the destination is disconnected.

Note that if the graph is not planar, face routing may fail. Figure 3 shows an example graph on which this pathology occurs. In this example, D is located physically in the interior of a face, but is only connected to the rest of the network graph by an edge that crosses this enclosing face. Face routing walks successive faces cut by the line from S to D, until it reaches the face enclosing D, whose first edge crosses line segment $\overline{SD}$ at point p. The right-hand rule then tours this face in its entirety, but fails to find an edge that crosses line segment $\overline{SD}$ at a point closer to D than p. Thus, face routing fails.

Wireless networks' connectivity graphs typically contain many crossing edges. A method for obtaining a planar subgraph of a wireless network graph is thus needed; greedy routing operates on the full network graph, but to work correctly, face routing must operate on a planar

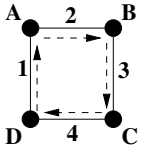


Figure 1: Right-hand rule. A sweeps counterclockwise from link 1 to find link 2, forwards to B, *&c.*

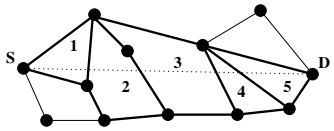


Figure 2: The faces progressively closer from S to D along line segment $\overline{SD}$, numbered in the order visited. Faces cut by $\overline{SD}$ are bordered in bold.

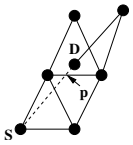


Figure 3: Example of face routing failure on non-planar graphs. There is no point closer to D than p on the face enclosing D.

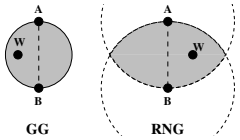


Figure 4: Definitions of the GG and RNG. A witness must fall within the shaded circle (GG) or lune (RNG) for edge (A,B) to be eliminated in the planar graph.

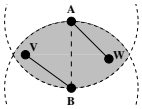


Figure 5: The RNG partitions a non-unit graph; edge (A,B) is eliminated.

subgraph of the full network graph. What is required is a *planarization* technique that is simply implementable with an asynchronous distributed algorithm.

Geographic routing algorithms planarize graphs using two planar graph constructs that meet that requirement: the Relative Neighborhood Graph (RNG) [26] and the Gabriel Graph (GG) [5]. The RNG and GG give rules for how to connect vertices placed in a plane with edges based purely on the positions of each vertex's single-hop neighbors. Both the RNG and GG provably yield a connected, planar graph so long as the connectivity between nodes obeys the *unit graph assumption:* for any two vertices A and B, those two vertices *must* be connected by an edge if they are less or equal to some threshold distance d apart, but *must not* be connected by an edge if they are greater than d apart. We shall refer to d as the *nominal radio range* in a wireless network; the notion is that all nodes have perfectly circular radio ranges of radius d, centered at their own positions.

The unit graph assumption is quite intuitive for wireless networks. The simplest ideal radio model is one where all transmitters radiate fixed transmission power perfectly omnidirectionally; receivers can discern all transmissions properly when they are received with above some threshold signal-to-noise ratio; and radio transmissions propagate in free space, such that their energy dissipates as the square of distance. Under that idealized model, there indeed exists a nominal radio range.

We briefly state the definitions of the GG and RNG, as we shall refer to them repeatedly in Section 3. The planarization process runs on a *full graph*, which includes *all* links in the radio network, and produces a *planar subgraph* of the full graph. We assume that each node in the network knows its single-hop neighbors' positions; such neighbor information is trivially obtained if each node periodically transmits broadcast packets containing its own position. Consider an edge in the full graph between two nodes A and B. Both A and B must decide whether to keep the edge between them in the planar graph, or eliminate it in the planar graph. Without loss of generality, consider node A. Both for the GG and RNG, node A searches its single-hop neighbor list for any *witness* node W that lies within a particular geometric region. If one or more witnesses are found, the edge (A,B) is eliminated in the planar graph. If no witnesses are found, the edge (A,B) is kept in the planar graph. For the GG, the region where a witness must exist to eliminate the edge is the circle whose diameter is line segment $\overline{AB}$. For the RNG, this region is the *lune* defined by the intersection of the two circles centered at A and B, each with radius $|\overline{AB}|$. We show these two regions in Figure 4.

Under the unit graph assumption, it is known that for a clustering of points in the plane, the set of edges in the Euclidean minimum spanning tree over those points is a subset of the set of edges in the RNG [26]. The edges in the RNG are in turn a subset of those in the GG; the intuition for this relationship lies in the relative sizes of the lune and circle regions. Finally, the set of edges in the GG is a subset of that in the Delaunay triangulation over the set of points [25]. These relationships dictate that the GG and RNG are both connected (so eliminating crossing edges cannot disconnect the network!) and planar, as desired. Note that if the network graph *violates* the unit graph assumption, the RNG and GG can produce a *partitioned* planarized graph [11], one that contains unidirectional links, and even one that is not planar. An example of a partitioning for the RNG appears in Figure 5. Here, there is no link between A and V, and none between B and W, though these links are shorter than the nominal radio range. Nodes A and B see witnesses W and V, respectively, though neither witness provides transitive connectivity. Both A and B conclude they should remove edge (A,B) in the planarized graph, and a partition results. Similar cases are possible in the GG.

We observe that whether radio graphs conform to the unit-graph assumption is a question of great importance, as partitioning the planarized graph used in face routing will cause routing failures. In the next section, we explore in detail the many reasons real radios violate the unit graph assumption, and give detailed examples of the pathologies these violations create in the GG and RNG.

Recently, Kuhn *et al.* have investigated relaxing the unit-graph assumption to improve the robustness of the GG planarization [18]. In the *Quasi-Unit Disk Graph* they propose, the nominal radio range is normalized to 1. Links *may not* exist between nodes greater than distance 1 apart, and links *must* exist between nodes less than a parameter d apart. For nodes between d and 1 distance apart, links may or may not exist; it's in this region where Quasi-Unit Disk Graphs are a more general class than unit graphs. Kuhn *et al.* provide an algorithm for replacing "missing" links between d and 1 in length with *virtual* links, that are essentially tunnels through multiple existing links. They show that the GG planarization succeeds on this augmented graph without partitioning it. Their analysis shows that this technique is only scalable when $d \geq 1/\sqrt{2}$; for lesser values of d (for which the unit-graph assumption is progressively relaxed further) virtual links may be comprised of increasingly long paths of physical hops.

3 Pathologies in Real Deployments

In the previous section, we demonstrated two situations where GPSR's perimeter-mode routing may fail: when crossing links remain after planarization is applied, and when planarization partitions the network graph. It is natural to ask how prevalent these pathologies are in real deployments of GPSR: are they so rare as to be of purely theoretical interest, or do they significantly negatively affect reachability between pairs of nodes? We confirm in this section that the latter is the case, using measurements taken on real wireless networks.

GPSR Implementation and Testbeds

We implemented GPSR for Mica-2 sensor motes. Our full-fledged nesC [8] implementation includes the GG and the RNG planarization algorithms (chosen via a configuration parameter), as well as greedy- and perimeter-mode packet forwarding. It also includes a hop-by-hop retransmission mechanism, as the default Mica-2 MAC layer does not implement link-layer retransmission. Finally, our implementation rejects wireless links whose quality—measured by probing link loss rate—is below a configurable threshold. This mechanism incorporates hysteresis to avoid oscillatory behavior on links whose quality is near the threshold. Our complete implementation is over 4500 lines of nesC code.

We measured this implementation's behavior on two testbeds. Each consists of Mica-2 motes that span a floor of an office building: one with 75 motes in Berkeley's Soda Hall, where offices are separated by floor-to-ceiling walls, and one with 51 motes at Intel Research Berkeley, where cubicles are separated by low dividers. We report only the Soda Hall results in the interest of brevity.[6]

Motes instrument most offices and some of the hallways in Soda Hall. Because the testbed is shared, we were able to use only a 50-node subset of it. As we could not control the placement of these devices, the GPSR failures discussed below are not contrived by careful node placement. We did, however, have one tool for controlling network topology: radio transmit power. At the default power setting on the testbed, all nodes were within two hops of each other. To generate an interesting multi-hop topology, we reduced the radio transmit power from 15 to 2. In the resulting topology, the average path length was around 5 hops, and the average node degree was 5.2. Note that controlling transmit power is roughly equivalent to appropriately scaling the geographic dimensions of the testbed. Finally, we statically configured nodes with their locations.

Pathologies

Figure 6 depicts the full network topology on the 50-node Soda Hall testbed, as is used by GPSR's greedy-mode forwarding. Our GPSR implementation does not forward on links with packet loss rates in excess of 30%; those links are not shown in the figure. Many links cross one another, particularly in the dense region of the network toward the left. It is the job of GPSR's planarization to eliminate these crossing links, to produce a planar graph for use by GPSR's perimeter-mode forwarding.

We measure the fraction of all pairs of nodes on this network that can reach one another with GPSR routing. In these measurements, we iterate over all nodes in the network, allowing one node at a time to send traffic to each other node in the network. We send 10 packets, and retransmit at the link level. If one or more packets reach the destination, we count that directed pair of nodes as connected, and in this way, measure routing algorithm success rather than short-term packet loss characteristics. We find that only 68.2% of directed node pairs can communicate successfully in the testbed—a significant fraction of node pairs experience *permanent partition!*

To help elucidate the reasons for these routing failures, we present in Figure 7 the network subgraph that results after our GPSR implementation distributedly applies the GG planarization to the full topology. There are three classes of pathology present in this network subgraph:

Network partitions: While the full network is connected, there are two connected components in Figure 7; the majority of the network comprises one connected component, and the nodes at the lower left of the figure the other. Such cases arise in situations such as those

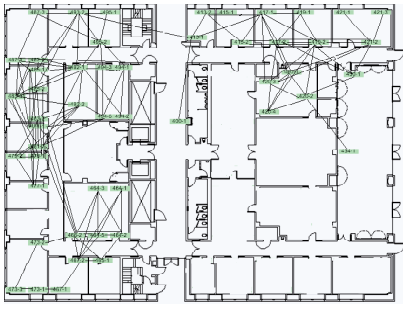


Figure 6: 50-node testbed. Links with packet loss rates over 30% are not shown.

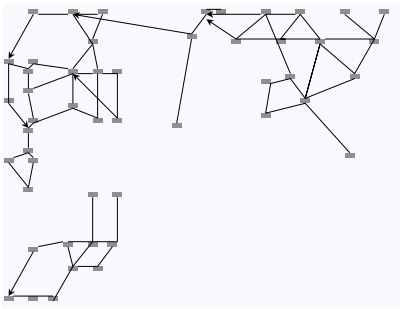


Figure 7: GPSR's GG subgraph on the 50-node testbed.

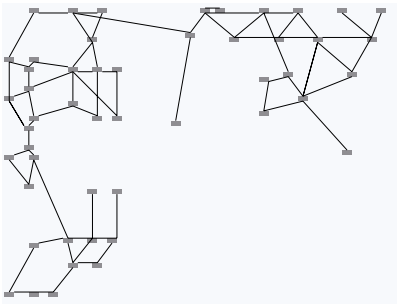


Figure 8: GPSR's GG/MW subgraph on the 50-node testbed.

previously described in Figure 5.

Asymmetric links: Links denoted with an arrow exist in the planar subgraph *only in the direction indicated.* Such links may give rise to unidirectional partitions in the planar subgraph, where an asymmetric link represents the only connectivity between two connected components. The GG and RNG planarizations produce asymmetric links in cases similar to that depicted in Figure 5; consider the case where W is not present in the graph. On that topology, $A \rightarrow B$ will remain, but $B \rightarrow A$ will not.

Crossing links: There are a few instances of crossing links that remain in Figure 7. For example, consider the long horizontal link that spans the hallway, and crosses a far shorter link. The GG and RNG planarizations may produce such pathologies when there are highly irregular radio ranges, as is the case here: the node at the right end of the long link cannot see any witnesses, and thus will not remove the long link; nor do the nodes at either end of the short, vertical link see any witnesses.

Radio range irregularities, which may be exacerbated by elimination of high-loss links, thus cause significant routing failures for GPSR in a real deployment. We expect other variants of GPSR to behave similarly, since they all use planarization methods based on unit-disk graphs. For context, we note that several measurement studies [1, 6, 27] have documented non-ideal radio behavior; however, ours is the first to quantify their impact on existing geographic routing protocols.

We have also implemented and experimented with a previously proposed fix to the GG's and RNG's tendency to partition graphs when radio ranges are irregular. The fix in question is the *mutual witness* (MW) extension to GPSR [11, 12, 24]. When node A considers whether to keep link (A,B) from the full graph in the RNG or GG planar graph, mutual witness dictates that A only eliminate link (A,B) if there exists at least one witness in the RNG or GG region that is visible *both* to A and B. This fact may be directly verified with local communication: if all nodes broadcast their neighbor lists (only a single hop), then all nodes may verify whether a particular neighbor shares a particular other neighbor. The intuition for this mutual witness is that it preserves connectivity: links are only eliminated in the planar graph if a transitive path through a witness is explicitly verified, rather than relying on the location of the witness to assure such a transitive path's existence. Unfortunately, MW suffers from another ill; on some non-unit graphs, it will *leave crossing links* in the graph produced by the RNG and GG. Indeed, in our experiments with MW, we observed this behavior: GPSR augmented with MW enables connectivity between only 87.8% node pairs in one experiment, leaving more than 10% of node pairs persistently disconnected. Figure 8 shows the subgraph the MW extension generates in this experiment; note the crossing edges that remain that give rise to routing failures.

In sum, these results suggest that current geographic routing protocols are impractical. Although we have demonstrated this only using relatively unsophisticated Mica-2 radios, we believe our conclusions hold for other kinds of wireless devices as well, since the failure of the unit-disk assumption as a result of obstacles or multipathing is fairly fundamental. We spend the rest of the paper discussing a qualitatively different and practicable approach to geographic routing. As an aside, we note that while many of the pathologies we describe above are caused by radio range irregularities, localization errors can also cause the same pathologies [14, 24]. We leave measurement of the effects of localization errors in testbed deployments to future work.

4 Cross-Link Detection Protocol

We have established that existing planarization techniques frequently cause face routing to fail on real wireless networks, where the unit-graph assumption is violated. We now proceed to describe the Cross-Link Detection Protocol (CLDP), a planarization technique that cannot cause face routing to fail on any connected graph. As such, CLDP is also robust to arbitrary localization errors [14]; we omit a detailed discussion herein for lack of space.

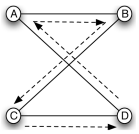


Figure 9: CLDP Probing using right-hand-rule, Case 1.

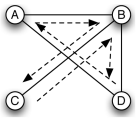


Figure 10: ..., Case 2.

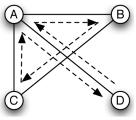


Figure 11: ..., Case 3.

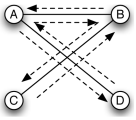


Figure 12: ..., Case 4.

4.1 CLDP Overview

To describe the essential ideas behind CLDP, we first consider a static graph consisting of several nodes and links. We make no assumptions about the connectivity of this graph (*i.e.,* to which other nodes a given node may be connected). However, we assume that nodes in the graph are assigned positions in some 2-dimensional coordinate system, that the graph is connected, and that all the links are bi-directional. Initially, we also make several other idealized assumptions (like link-serialized execution of the protocol) to simplify exposition. We will return a bit later to consider the applicability of CLDP to more realistic wireless networks: in particular, we will consider the impact of node and link dynamics, and present a truly distributed, parallel realization of CLDP. We do not explicitly consider node mobility in our evaluation of CLDP, and leave that to future work.[7]

The high-level idea behind CLDP is simple: each node, in an entirely distributed fashion, *probes* each of its links to see if it is *crossed* (in a geographic sense) by one or more other links. A probe initially contains the locations of the endpoints of the link being probed, and traverses the graph using the right-hand rule. For example, in Figure 9, consider a probe originated by node D for the link (D,A). It contains the geographic coordinates of D and A, and traverses the graph using the right-hand rule, as shown by the dashed arrows. When the probe is about to traverse the link (B,C), node B "notices" that this traversal would cross (D,A); B records this fact in the probe so that when the probe returns to D, D notices a cross-link and "removes" either the (A,D) link or the (B,C) link (after a message exchange with B). By symmetry, the cross-links would have been detected by a probe of (A,D) originated by A or a probe of (B,C) originated either by B or C.

Care must be taken in dealing with degenerate crossings caused by exactly colinear links. A correct way to deal with these is to randomly, but slightly, perturb the reported location of each node to make the likelihood of such links vanishingly small. To simplify our discussion, we ignore such degeneracies in the rest of this paper.

We have described CLDP in a decentralized fashion, but to understand CLDP's properties, it helps to envision the results of applying CLDP on all links of a static (*i.e.,* unchanging), arbitrary (*i.e.,* no specific connectivity assumptions), connected graph. Initially, assume that all the links in this graph are marked *routable*. Then, suppose that each link is probed repeatedly and in some order with the constraint that only one probe is active at any given time (this is an idealization we relax later). As we have described above, a probe may cause a link to be removed. When we say CLDP "removes" a link, we mean that the link is marked *non-routable*. The set of routable links forms a *routable subgraph*. Furthermore, *all CLDP probes traverse the current snapshot of the routable subgraph*. Cross-links are not always marked non-routable; we show later how CLDP preserves cross-links the deletion of which would render the routable subgraph disconnected. This property implies that if the graph is connected to start with, CLDP does not partition it. The probing stops when subsequent probing of links would not cause any link to be marked non-routable.

We say a graph is *safe* if face routing between all pairs of nodes in the graph is guaranteed not to fail. As we discuss in Section 4.5 (and our simulations and experiments Section 5 bear this out as well), CLDP always produces a safe routable subgraph from any arbitrary input connected graph. This result is surprising for the following reason. It is easy to see that CLDP attempts to planarize the routable subgraph by removing cross-links, and face routing is known not to fail on a planarized graph. However, there is no *a priori* reason to believe (and no prior literature that suggests) that using the right-hand rule repeatedly to detect and remove cross-links will always result in a planarization (modulo the cross-links that need to be preserved to avoid disconnections) on an arbitrary graph.

As a practical matter, other forwarding strategies also work perfectly on the CLDP-derived routable subgraphs, such as GPSR's combination of greedy- and perimeter-mode traversals [13], and GOAFR's improvement that uses ellipses to bound face traversals when possible [17]. Note further that greedy forwarding uses the full graph (including links marked "non-routable" by CLDP); only face routing uses the CLDP-derived routable subgraph during recovery from local maxima.

In describing CLDP, we have made two simplifying assumptions: strictly sequential probing of links, and no node or link dynamics. In the following sub-sections we relax these two assumptions. Before doing so, however, we consider two other problems: how CLDP deals with cross-links whose removal would partition the routable subgraph, and how CLDP detects multiple cross-links.

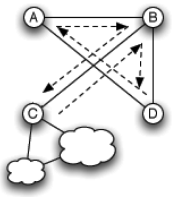


Figure 13: Effect of "clouds" on probes.

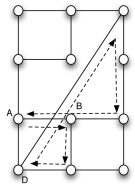


Figure 14: Routable subgraph depends on probe ordering.

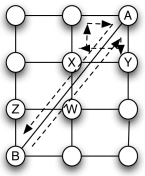


Figure 15: Multiple Cross-Links.

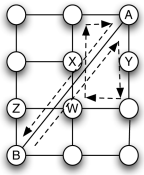


Figure 16: Repeated CLDP probes.

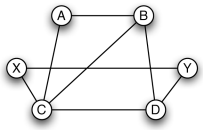


Figure 17: Probing a link may not detect a cross-link.

4.2 Partitions in the Routable Subgraph

In Figure 10, the removal of the (B,C) link would disconnect C from the rest of the network. Similarly, in Figure 11, the removal of the (A,D) link would disconnect D, and in Figure 12 the removal of either crossing link would partition the network.

To understand how CLDP deals with this situation, examine the paths taken by the CLDP probes originated by D in each of the figures (by symmetry, one can make similar observations about probes initiated by C). Notice that in every case, when disconnecting a crossing link would partition the graph, the CLDP probe traverses that link *once in each direction*. In Figure 11, for example, the CLDP probe returns to D over the link on which it was sent (*i.e.,* the (A,D) link). Intuitively, it is clear why this should be so: there is no closed face over which the probe can return. In Figure 10, the CLDP probe originated by D traverses link (B,C) once in each direction. From this, B (or C) can infer that removing link (B,C) would cause a partition.

While we have given the simplest possible examples, our observations generalize easily to arbitrary topologies attached to the "non-removable" link. For example, if in Figure 10, node C were connected to many "clouds" (Figure 13), the CLDP probe would return on the (B,C) link. Thus, when a CLDP probe traverses either the link being probed (or its cross-link) in both directions, CLDP infers that removal of that link could disconnect the routable subgraph, and does not remove the link. By this rule, CLDP would mark both the (A,D) and the (B,C) links in Figure 12 routable. We point out an important property of the routable subgraphs derived by applying CLDP—*they may contain crossing links*.

Thus, the correct rule for marking links non-routable can be stated as follows. Suppose any node N probes an attached link L and finds a cross-link L':

Case 1: If both L and L' can be removed (*i.e.,* the CLDP probe traversed neither link twice), remove L.

Case 2: If L can be removed, but L' cannot, remove L.

Case 3: If L cannot be removed, but L' can, signal the appropriate nodes to remove L'.

Case 4: If neither link can be removed, do nothing.

Consider the application of this rule to the network in Figure 14, which illuminates an important property of CLDP: that *different* routable sub-graphs may be generated by applying CLDP to the same graph, depending upon the order in which links are probed. For example, if (A,B) were probed first, then (C,D) would be removed, and vice versa.

4.3 Multiple Cross-Links

Thus far in our discussions, we have assumed that a link is crossed by at most one other link. But consider the situation depicted in Figure 15 where a long link (A,B) is crossed by three other links. In arbitrary graphs, of course, this situation will not be uncommon.

CLDP generalizes rather easily to this case. Repeatedly probing a link until no removable cross-links are found will keep the resulting routable sub-graph safe. Consider Figure 15 and assume that B probes link (A,B). The first such probe will traverse the faces shown, detecting the cross-link (X,Y), which will be removed. A second probe sent by B (Figure 16) will detect the (X,W) cross-link, resulting in the removal of that link (and so on). Our examples of multiple cross-links are a bit misleading, as they suggest that repeatedly probing a link will detect *all* cross-links. This is not, in general, true: probing *one* of a pair of cross-links is not guaranteed to find the crossing (intuitively, that link may be obscured by other, perhaps non-removable) links. The other link may also have to be probed (from both ends) before the cross-link is detected. Consider, for example, the topology in Figure 17. In this topology, CLDP probes from either end of the (B,C) link are confined to the adjoining triangles, and are unable to detect the (X,Y) link. The (B,C) cross-link is only detected after repeatedly probing the (X,Y) link.

4.4 Concurrent Probing

Thus far, we have assumed that CLDP probes are *serialized*. However, this kind of global serialization is unachievable without significant messaging cost in large networks. A design that permits nodes to probe links concurrently is clearly more desirable.

Unfortunately, concurrent probing can render the routing subgraph disconnected. Consider Figure 9 and assume that while D probes link (A,D), C concurrently probes link (B,C). When each probe returns, C and D each detect a cross-link, and mark their directly attached links non-routable (assume that either link can be removed), leaving the routable subgraph disconnected. Such a race condition can be prevented using a simple *tie-break* rule that deterministically decides which cross-link should be deleted. However, the tie-break rule does not guarantee correctness in the general case.

A simple approach would be to *lock* a link while it is being probed. CLDP drops probes that encounter a locked link in either direction, and retries them later. This approach effectively ensures that the faces adjoining the locked link are not altered while the link is locked (modulo changes caused by node failures or additions, which we discuss later).

CLDP uses this basic strategy, but takes care to avoid race conditions in cases where the cross-link (and not the probed link) must be removed. Furthermore, it reduces convergence time using a few simple optimizations, since the basic strategy can cause many dropped probes. Finally, it also reduces probing overhead by avoiding probes on links which have already been determined to be routable, unless one of the adjoining faces has changed. We now describe these modifications.

First, CLDP uses *lazy* locking. That is, when CLDP needs to probe a link, it *first* sends a probe without locking the link. If this probe returns indicating either that there are no cross-links or that this link and its cross-link cannot be removed (Case 4, Figure 12), CLDP marks the link to be routable. Thus, in this case (which one expects to be common for small faces on dense networks), CLDP converges quickly without locking links. Routable links are marked *dormant* and not subsequently probed unless woken up; we later describe how this happens.

There are two other possible outcomes of a probe message; either the probed link needs to be removed from the CLDP-derived graph (*e.g.,* Case 2, Figure 10), or its cross link needs to be removed (*e.g.,* Case 3, Figure 11). In the former case, CLDP enters a *commit* phase, where it locks the probed link, and re-probes the link but using a specially marked "commit" message. All probes traversing a locked link in either direction are dropped. However, when a commit message traverses a locked link, a deterministic tie-break is applied which ensures that if two links on the same face are being "commit"-ed simultaneously, only one of the commit messages succeeds in traversing the face. When the "commit" probe succeeds, CLDP unlocks the probed link, and marks it as *non-routable*. The act of marking a link non-routable changes the faces adjacent to the link. As Figures 15 and 16 show, removal of a link can reveal cross-links (*e.g.,* the (X,W) link does not see the (A,B) cross-link until the (X,Y) link is removed from the graph). Accordingly, the changed faces must be re-probed. To accomplish this, when CLDP removes a link (*i.e.,* marks it non-routable), it awakens the two adjacent *dormant* (see above) links (*i.e.,* those obtained by applying the right-hand rule and the left-hand rule from this link).

The last case to consider is when a probe indicates that the cross-link (*e.g.,* link (B,C), Figure 11) must be removed. Recall (Figure 17) that, in general, a probe of the cross-link might not reveal the crossing. For this reason, when a probe indicates the cross-link needs to be removed, CLDP walks the corresponding face again using a "commit" probe, and locks the cross-link after the probe reaches it. When that probe succeeds, the node notifies both ends of the cross-link to mark the link non-routable.

Finally, we describe CLDP's behavior when a link is added to or deleted from the underlying connectivity graph. When a link is added to the underlying graph, CLDP awakens the adjacent dormant links. This causes links on the corresponding faces to be probed again, eliminating cross-links when necessary. Link deletion presents a more subtle problem. Consider Figure 15, and suppose that links (Z,W) and (X,Y) have been marked non-routable. Now, suppose that link (A,B) fails. The simplest way to restore the links (Z,W) and (X,Y) to the routable sub-graph would be to periodically re-probe these links. This is what CLDP does. It is possible to design optimizations that can reduce the overhead of periodic probing. For example, node A could remember which cross-links were removed when (A,B) was probed, and notify the ends of those cross links when (A,B) fails. We have left the design of these optimizations for future work.

CLDP implements its probing actions using a simple state machine and a protocol consisting of several message types. In the interest of brevity, we refer the interested reader to [14] for a detailed specification of the CLDP protocol.

4.5 Statement of Correctness

Space constraints limit us only to state the theorems that prove CLDP's correctness. In this formal analysis, we assume that the full network graphs are static and have no degeneracies: no vertices are coincident, and no pairs of edges at a single node have the same incident bearing; there is a provably correct way to handle the latter degeneracy, elided because of space constraints. Thus, the notion of a "crossing" is well-defined. For each graph define a (perhaps empty) set of crossings C; each element of C is a pair of edges that intersect in the plane.

Our results are based on the fact that all face walks eventually return to their starting points. We use the

following terminology to describe how a face walk returns to its starting point. An edge is *singly-walked* if a face walk starting on that edge does not return via that same edge (in the opposite direction). An edge is *doubly-walked* if it returns via the same edge in the opposite direction. The general rule in CLDP is that when a crossing is detected, no doubly-walked edge can be removed, but if one of the crossing edges is singly-walked, then an edge is removed. Our first result is a general observation about crossings in connected graphs.

Theorem 4.1 *If a connected graph G has at least one crossing, then there is at least one face with a crossing.*

This result shows that if we had used a version of CLDP that eliminated *all* crossings then we would end up with a set of connected planar components. To help state our next result, we term a graph *CLDP-stable* if CLDP would not eliminate any edge in the graph, were the edges probed in serial fashion. We then have:

Theorem 4.2 *Geographic routing never fails on a connected CLDP-stable graph.*

This says that if we use CLDP's rules about when to eliminate crossings, then we end up with a connected graph on which one can reliably use geographic routing.

5 Simulation Results

The above theorems assert CLDP's correctness on static graphs. However, to show that CLDP is practical on real wireless networks, we examine the performance of CLDP through simulation in this section, and through experimentation in the next.

Methodology and Metrics We implemented CLDP (and other geographic routing protocols, described below) in TinyOS [10], the event-driven operating system used on the Mica-2 motes. TinyOS code can be directly executed on TOSSIM [19], a process-level simulator that can be used to directly debug and evaluate sensor network applications and protocols. Our implementation of CLDP in TinyOS is 750 lines of nesC code. In this section, we report simulation results obtained from running CLDP and other protocols using TOSSIM's support for packet-level simulation.

In this section, we compare (whenever appropriate) CLDP's performance against three alternatives, *GPSR* denotes the full implementation of GPSR using the Gabriel Graph for planarization, greedy forwarding, and perimeter traversal for routing around voids. We use *GPSR* to provide context for CLDP's performance. *GPSR′NOPLAN* denotes a protocol that forwards packets using GPSR on the full connectivity graph (*i.e., without* planarization). *GPSR′NOPLAN* delineates the baseline performance of face walking on the networks we study. *GPSR′GG/MW* includes, in addition to GPSR and planarization, an implementation of the "mutual witness" procedure for avoiding unidirectional links and disconnections in the planarized graph when the unit-graph assumptions are violated (Section 3). *GPSR′GG/MW* quantifies the inadequacy of that proposed fix for planarization failures, thereby highlighting the need for CLDP. *GPSR′CLDP* denotes our proposed protocol using CLDP, greedy forwarding, and perimeter traversal.

In each of our simulations, we use a 200-node topology in which nodes are randomly positioned on a fixed-size two-dimensional surface. We conducted simulations on two types of networks: wireless networks with an idealized radio model with circular radio ranges (we introduce reality in the form of obstacles), and Bernoulli random graphs which have a fixed connection probability for any pair of nodes, regardless of Euclidean distance between the nodes. For our wireless network simulations, we evaluate the performance of various geographic routing protocols as a function of node density. Our measure of density is the average number of neighbors of a node. We scale the area of the surface in order to vary node density; for our highest density we use an area of 1300 x 1300 units, while for our lowest, we use an area of 2000 x 2000 units. The radio range is 180 units.

In our simulations with obstacles, the number of obstacles is indicated by a parameter f, such that fN is the total number of obstacles (N is the number of nodes). Each obstacle is of fixed length (45 units) in each of our simulations. The mid-point of the obstacle is randomly positioned on the two-dimensional surface, and the orientation of the obstacle is equally likely to be either vertical or horizontal. This obstacle model helps us stress CLDP and other protocols to varying extents in order to measure their performance.

Our Bernoulli random graphs are generated in the obvious way: we flip a weighted coin for each pair of nodes, assigning a link between them with the desired connection probability.

For each simulation we first generate a network topology. We then ensure that the topology is connected. At the beginning of the simulation, TOSSIM enforces a boot-up time during which nodes are started randomly. In our simulations, 200 nodes are started randomly in the first 30 seconds. Following the boot phase, each simulation consists of two phases. In the first phase, we let the appropriate routability determination protocol (CLDP, or GPSR's planarization and/or mutual witness procedure) execute at each node long enough for the network to converge. In the second phase, we send packets pairwise bidirectionally between nodes in a staggered manner to minimize wireless collisions. This latter phase tests for routing failures. For each data point in the graphs below, we run 50 random topologies. We have verified that this is sufficient to produce negligible 95% confidence inter-

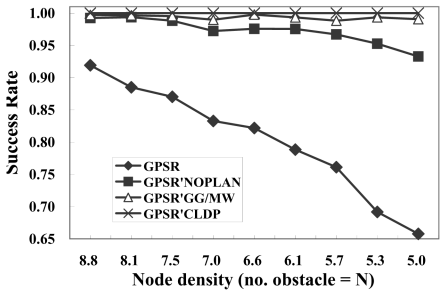


Figure 18: Success rate for 1.0N obstacles.

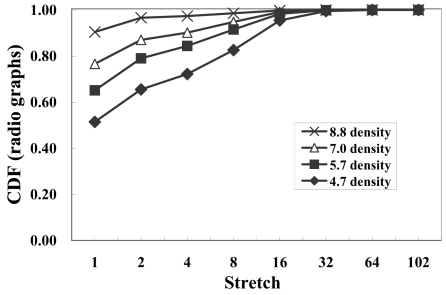


Figure 19: CDF of stretch (1.0N obstacles).

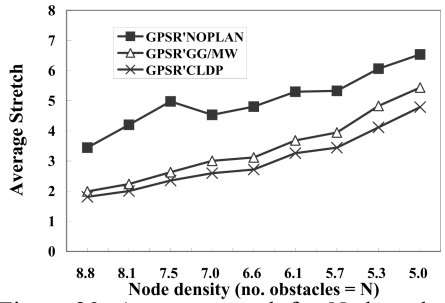


Figure 20: Average stretch for N obstacles.

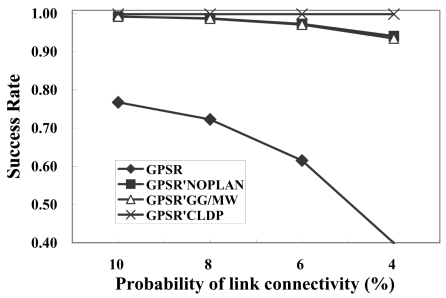


Figure 21: Random graph success rate.

vals for the mean values of our metrics.

We do not simulate packet losses due to interference or buffer overrun in either phase. Our simulations do drop packets, however, when face routing fails. Packet losses would increase the convergence time of CLDP, or would alter the level of concurrent probing in CLDP. Our simulation methodology already introduces significant concurrency by ensuring that all nodes start at nearly the same time. (Note that our testbed measurements include interference and buffer overrun effects, of course.)

We use two primary measures of performance. The *success rate* measures the fraction of sender/receiver pairs for which packet transmissions from the sender are successfully received. The *average stretch* measures the average of path stretch for all sender/receiver pairs. The stretch of a path is the ratio of the number of hops using the routing scheme in question to the number of hops in the shortest path. We also evaluate the overhead and convergence time of CLDP; we define these metrics below.

Given space constraints, we only present a sampling of simulation results extensively described in [14]. In particular, we omit results validating CLDP's correctness on networks with localization errors as well as a detailed discussion of CLDP's performance on random graphs.

Wireless Networks with Obstacles Figure 18 shows the success rate as a function of node density for our various protocols, in the presence of N obstacles. Note that this is an extremely harsh environment, with as many obstacles as nodes. As expected, CLDP allows perfect delivery success across all node densities we evaluated. Interestingly, GPSR's planarization procedure fails rather dramatically in the presence of even a moderate number of obstacles. In these circumstances, it appears to be more advantageous simply to use GPSR on the connectivity graph without planarization. The mutual-witness procedure fixes many of GPSR's shortcomings and is close to perfect in some cases. At most densities it can establish paths between 99% or more node pairs, but it is never perfect. In a real deployment, however, MW fails far more dramatically, as discussed in Section 3.

Figure 20 plots the average stretch as a function of node density for our various protocols, in the presence of N obstacles. CLDP exhibits an average stretch between 2 and slightly above 4, with a higher stretch at lower densities. CLDP outperforms GPSR'GG/MW in this respect; CLDP removes only cross links, but GPSR'GG/MW removes all links that are witnessed by planarization and hence incurs higher stretch. However, CLDP may exhibit long paths. This is evident from the CDF of stretch for CLDP (Figure 19, with N obstacles). Notice the long tail of the distribution, in which some paths have a stretch of over 100! Across the range of densities we explore, though, 60-95% of the paths have a stretch less than 2.

Random Graphs To stress CLDP, we also simulated it on Bernoulli random graphs with various connectivity probabilities. As Figure 21 shows, CLDP exhibits no routing failures, even on random graphs. By contrast, all other variants exhibit significant routing failures on sparse random graphs (low connection probabilities). In particular, MWP exhibits more systematic routing failures than on wireless networks. Clearly, none of these other protocols is practical for routing on random graphs.

Overhead We measured how many CLDP messages are needed to add a link to a wireless network with N ob-

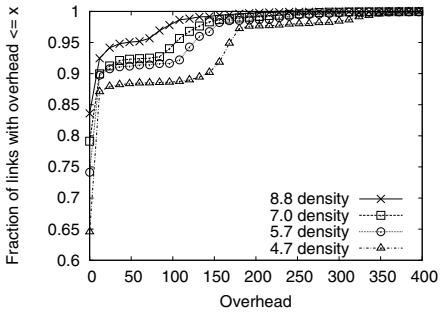


Figure 22: Overhead for wireless network with N obstacles.

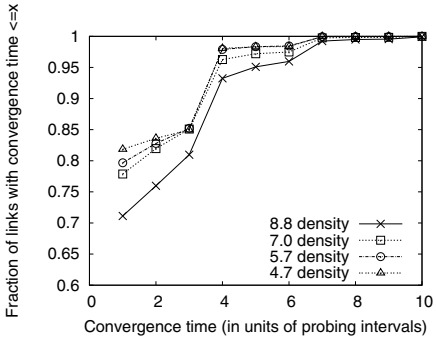


Figure 23: Convergence time distribution for wireless network with N obstacles.

stacles. This gives us some idea of the overhead incurred by CLDP. In our experiments for measuring overhead, after a network has reached steady state, two nodes not directly connected to each other are randomly selected and an additional link between them is activated.

The *overhead* is the total number of CLDP control messages (probe and commit) traversing a link in either direction until the network has converged. Figure 22 plots the distribution of link overheads averaged over 20 link additions on each of 200 wireless topologies. It shows that about 85%-90% of links see fewer than 4 messages, but a very small fraction of links see upwards of 100 messages. This latter phenomenon can be explained as follows. Assume that a new link is added which crosses existing edges. When CLDP removes these crossing edges, it needs to wake up all links on the faces adjacent to the removed link in order to detect successively hidden cross-edges. These links generate probe messages to see if they are crossed by others. Hence, the number of messages observed on a link depends on the size of the face. Clearly, in our wireless topologies (particular in the ones with lower density), there exist long faces.

Network Convergence Time We measured how quickly CLDP converges both on wireless networks with N obstacles and on Bernoulli random graphs. In experiments of convergence time, 200 nodes are initially started roughly simultaneously. In our CLDP implementation, nodes periodically probe their attached links before the links become dormant. Thus, the convergence time of CLDP is a function of this periodic timer. The *convergence time* of a link is defined as the number of CLDP probing intervals before a link becomes dormant and remains thus (Section 4.4). Notice that our experiments measure link convergence at *startup*; one would expect that in steady-state, the time for convergence after a single link failure and recovery can be expected to be considerably lower. Figure 23 shows the convergence time distribution for wireless networks with N obstacles. In Figure 23, about 95% of links converge within 4 probe intervals and all links converge within 9. In practice (Section 6), convergence times are slightly longer.

Network Dynamics Finally, we conducted simulations to evaluate CLDP's resilience to network dynamics. These experiments were done on 200 wireless networks with N obstacles as well as 200 Bernoulli random graphs. In all experiments, we took each given topology, randomly selected some links, and marked them non-routable in order to force those links to be re-probed by CLDP. Then we let CLDP execute at each node. Initially, these non-routable links are not used for CLDP probing. Over time, however, these links are woken up and are CLDP-probed. After all links had reached a dormant state, we determined whether packets could be routed between all pairs. In every case, CLDP converged to a network with 100% pairwise connectivity. Note that if a link flaps, CLDP will continuously attempt to probe the link. It might be possible to dampen this activity, but we have not investigated such mechanisms.

Summary In every simulation experiment, CLDP establishes routing paths between all node pairs. It exhibits reasonable stretch, overhead, and convergence times. Moreover, it works well under network dynamics. We next measure how CLDP performs on actual wireless testbeds.

6 Experimental Results

In this section, we describe CLDP's performance in deployment on wireless sensor network testbeds.

Testbeds and Experiments

We deployed CLDP on two different sensor node testbeds; as geographic routing's behavior is sensitive to the detailed placement of nodes and obstacles, we sought to demonstrate CLDP's behavior for multiple node and obstacle placements, to the extent possible using testbed resources at our disposal. The first testbed we shall label R, and consists of 75 Mica-2 "dots" with 433 MHz radios, deployed roughly one per room on one floor of Berkeley's Soda Hall. As described in Section 3, this was a shared testbed infrastructure, so we had no control

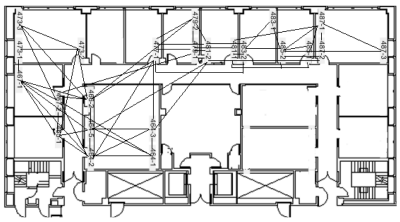


Figure 24: Node layout for *Rs*.

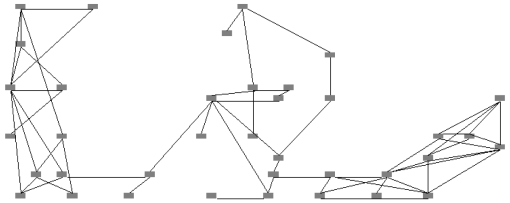


Figure 25: . . . for *C*.

over node layout and were able to use only a subset of the nodes for our experiments. We report performance measurements obtained on two different subsets of this testbed: *Rs* (Figure 24) which contains 23 nodes, and *Rm* (Figure 6) which contains 50 nodes.

The second testbed, which we shall call *C*, consists of 51 Mica-2 "dots" deployed across a floor of Intel Research Berkeley, of which we were able to use 36. In addition to environmental differences (cubicles in *C* *vs.* rooms in *R*), the testbeds differ in that *C*'s nodes are suspected to have poorer quality radios. Furthermore, *C*'s radios operate at 916 Mhz, and incur interference from other nearby devices in that unlicensed band. Again, on *C* we had no control over node layout.

As described in Section 3, in these testbeds we adjusted node transmit power to obtain a multi-hop topology. For *Rm* and *C*, notice that the topologies stress geographic routing protocols significantly–they contain two or more "clusters" of sensor nodes linked by one or two links, a configuration that triggers perimeter-mode routing frequently. Of course, such topologies aren't very practical since their capacity would be constrained by the bottleneck links. However, they can give some idea of worst-case CLDP performance, as we discuss below.

We thus conducted three sets of experiments: *Rm*, *Rs*, and *C*. In each experiment, nodes were configured with their locations. We started all nodes roughly simultaneously and let CLDP probing converge. We logged every packet (all devices in both testbeds had console access through a serial port), and we also recorded pair-wise link quality. In addition, for *Rs*, we conducted an experiment in which we sent 50 packets between each pair of nodes in order to measure packet delivery performance. Our packet forwarding implementation tries up to three link-layer retransmissions per hop.

Results

In this section, we report on the performance of CLDP according to a variety of metrics. At the outset, we point out that in all three experiments, CLDP was immune to the pathologies described in Section 3 and established pairwise connectivity between 100% of node pairs.

Path Performance One aspect of a routing protocol's path performance is stretch. For most node pairs (Figure 26), CLDP's stretch is reasonable (2 or 3). However, CLDP does exhibit fairly significant stretch (up to 20 in some cases) for a small fraction of node pairs. High stretch arises from long paths between pairs of nodes. Often, such long paths arise during traversal of the outer perimeter of the network.

One might argue that comparing CLDP paths with shortest paths is unrealistic, since shortest-path routing is known to offer low throughput [3] over a wireless network whose links span a wide range of packet delivery rates. For this reason, we measure the quality of CLDP's path selection. Figure 27 computes the distribution of pairwise packet delivery rates (the fraction of delivered packets) for both CLDP (measured on *Rs*) and ETX (computed from link quality estimates on *Rs*).[8] CLDP's packet delivery performance is comparable to, but slightly worse than this "idealized" ETX. A comparison with a real implementation of ETX might lessen the discrepancy between the two considerably. Finally, we note that ETX (when implemented on a proactive protocol like DSR, on a network with a dynamic topology) is likely to incur higher overhead than CLDP.

Convergence Time Figure 28 shows that most links converge within 15–20 probe intervals; with a 15 second probe timer, this corresponds to about 4.0 minutes. However, some links exhibit very long convergence times (up to 70 intervals). Our experiment measures *startup* convergence, since all the nodes are started roughly simultaneously. For CLDP, this is the worst case: when all links are simultaneously probed, link locking will delay convergence significantly. This also explains why *Rm* and *C* show a qualitatively different behavior; the bottleneck links between clusters induce significant probe contention.

A more realistic measure of link convergence time is the time it takes for a single link to converge when the rest of the network is in steady state. Even for a moderate size network, we couldn't automate this experiment easily, so we estimate this interval. We obtained our *estimated steady-state convergence time* by counting only those CLDP probes that do not encounter locked links. By this measure, CLDP converges very fast (Figure 29); more than 99% of the links converge within 6 intervals in all three experiments.

In addition, we also conducted an experiment where we started with a converged network, and manually disabled and then re-enabled an arbitrary link chosen from ten arbitrarily selected nodes. We then measured the time

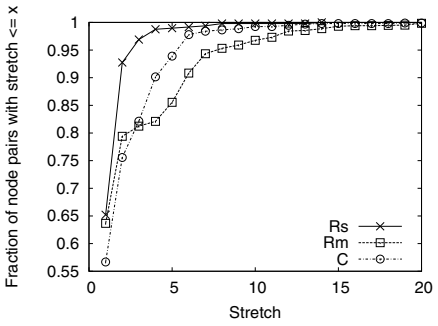


Figure 26: CDF of stretch.

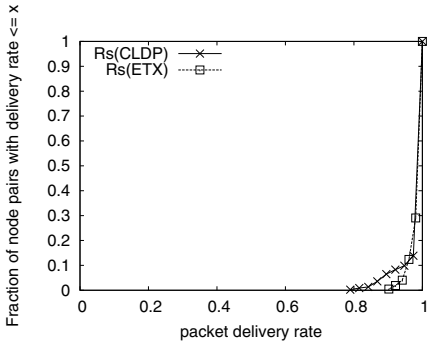


Figure 27: CDF of pairwise packet delivery rate.

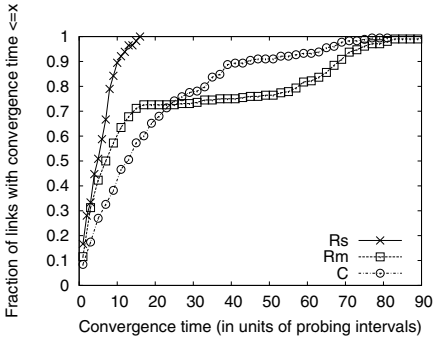


Figure 28: CDF of convergence time.

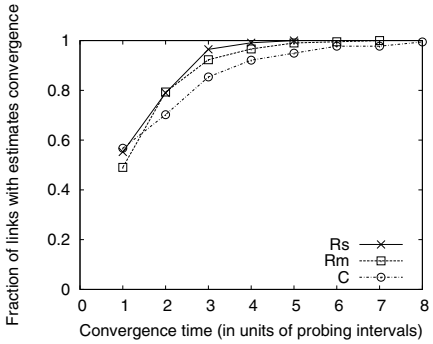


Figure 29: CDF of estimated steady-state convergence.

for CLDP to converge after each transition. After disabling a link, CLDP converged on average within 1.86 probing intervals; after enabling a link, CLDP converged on average within 0.59 probing intervals.[9]

Overhead Finally, we quantify the overhead of CLDP from our measurements. The primary metric we study is the distribution of probing overhead on individual links. However, rather than merely count the number of CLDP probe messages on each link for the entire duration of the experiment, we compute the average number of messages on each link[10] *per probing interval*. Normalizing the overhead this way helps us compare different experiments whose convergence times are different. Figure 30 shows that the overhead of CLDP is quite low; even on the busiest link, CLDP incurs less than one packet per second (if we assume a probe interval of 15 seconds), and on most links the overhead is significantly less.

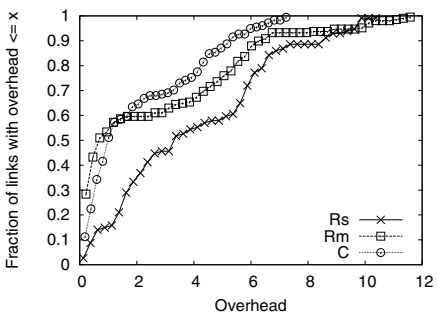


Figure 30: CDF of overhead.

7 Conclusion

We have motivated, described, and evaluated CLDP, which, to our knowledge, is the first distributed planarization protocol that renders geographic routing correct on arbitrary graphs. Simulations and measurements on real testbeds indicate that CLDP is quite practical: it offers high delivery rates, low overhead, and fast convergence. In future, we plan to investigate CLDP's overhead and robustness on more dynamic topologies, as well as the effect of localization errors on CLDP's path stretch in deployment.

Acknowledgments

We thank the anonymous reviewers for their comments, and Ellen Zegura for her thoughtful shepherding of this paper. We are further indebted to Om Gnawali for the comparison with ETX, and Jerry Zhao, Xin Li, and Wei Hong for helping us to use the UC Berkeley and Intel Research Berkeley sensor network testbeds.

Notes

[1]Other face routing techniques [17] can be used as well; CLDP preserves their correctness, but may affect their performance.

[2]For lack of space, we only present the resulting theorems, not their proofs, which may be found in [14].

[3]We note that there exist other routing algorithms that make use of position information, such as LAR [16], but we restrict the scope of our work to the family of face-routing algorithms in which a node forwards to a single neighbor on the basis of geographic information.

[4]We refer to links and edges interchangeably throughout the paper.

[5]Other face-change rules are possible, including changing faces at the edge whose crossing of $\overline{SD}$ is the *closest* such crossing to D on the

current face. We use the first crossing, not best crossing, throughout this paper; this choice is known to be average-case efficient, and has been refined [17] to be worst-case optimal.

[6]While pathologies in geographic routing are sensitive to the particular placement of nodes and the obstacles between them, we observed similar results on the two testbeds, and thus expect similar behavior in other real deployments.

[7]In principle, CLDP wouldn't need additional mechanisms to function under mobility, and would work well when link disconnections due to mobility occur on much longer timescales than the time required to complete CLDP probes.

[8]While CLDP uses only "good" links, our simulation of ETX is not similarly constrained.

[9]If a link is probed exactly once before it becomes dormant, that counts as a convergence time of zero.

[10]Although we count the number of messages on a link, recall that in our implementation, each message on a "link" constitutes a radio broadcast. Interpreted thus, our measure of overhead indicates the number of data packets that CLDP probing displaces in our deployment. A more general measure, and one that we have not investigated since it depends on deployment density and other environmental factors, is the fraction of transmission capacity that CLDP probing overhead occupies.

References

[1] S. Biswas and R. Morris. Opportunistic routing in multihop wireless networks. In *Proc. ACM HotNets Workshop*, Boston, MA, USA, Nov. 2003.

[2] P. Bose, P. Morin, I. Stojmenovic, and J. Urrutia. Routing with guaranteed delivery in ad hoc wireless networks. In *Proc. ACM DIALM Workshop*, pages 48–55, Seattle, WA, USA, Aug. 1999. ACM.

[3] D. De Couto, D. Aguayo, B. Chambers, and R. Morris. Performance of multihop wireless networks: Shortest path is not enough. In *Proc. ACM HotNets Workshop*, New Jersey, USA, Oct. 2002.

[4] G. Finn. Routing and addressing problems in large metropolitan-scale internetworks. Technical Report ISI/RR-87-180, USC/Information Sciences Institute, Mar. 1987.

[5] K. Gabriel and R. Sokal. A new statistical approach to geographic variation analysis. *Systematic Zoology*, 18:259–278, 1969.

[6] D. Ganesan, D. Estrin, A. Woo, D. Culler, B. Krishnamachari, and S. Wicker. Complex behavior at scale: An experimental study of low-power wireless sensor networks. Technical Report UCLA/CSD-TR-02-0013, University of California, Los Angeles, Computer Science Department, 2002.

[7] J. Gao, L. Guibas, J. Hershberger, L. Zhang, and A. Zhu. Geometric spanner for routing in mobile networks. In *Proc. ACM MobiHoc*, pages 45–55, Oct. 2001.

[8] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. ACM SIGPLAN PLDI*, San Diego, CA, June 2003.

[9] R. Gummadi, N. Kothari, Y.-J. Kim, R. Govindan, B. Karp, and S. Shenker. Reduced state routing in the Internet. In *Proc. ACM HotNets Workshop*, San Diego, USA, Nov. 2004.

[10] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proc. 9th ACM ASPLOS*, pages 93–104, Cambridge, MA, USA, Nov. 2000. ACM.

[11] B. Karp. *Geographic Routing for Wireless Networks*. PhD thesis, Harvard University, 2000.

[12] B. Karp. Challenges in geographic routing: Sparse networks, obstacles, and traffic provisioning. Presentation at the DIMACS Workshop on Pervasive Networking, May 2001.

[13] B. Karp and H. T. Kung. GPSR: Greedy perimeter stateless routing for wireless networks. In *Proc. ACM/IEEE MobiCom*, pages 243–254, Boston, Mass., USA, Aug. 2000. ACM.

[14] Y.-J. Kim, R. Govindan, B. Karp, and S. Shenker. Practical and robust geographic routing in wireless networks. Technical Report 04-832, Department of Computer Science, University of Southern California, 2004.

[15] L. Kleinrock and H. Takagi. Optimal transmission ranges for randomly distributed packet radio terminals. *IEEE Trans. Comm.*, 32(3):246–257, 1984.

[16] Y.-B. Ko and N. Vaidya. Location-aided routing in mobile ad hoc networks. In *Proc. ACM/IEEE MobiCom*, Aug. 1998.

[17] F. Kuhn, R. Wattenhofer, Y. Zhang, and A. Zollinger. Geometric ad-hoc routing: Of theory and practice. In *Proc. ACM PODC*, Boston, MA, USA, July 2003.

[18] F. Kuhn, R. Wattenhofer, and A. Zollinger. Ad-hoc networks beyond unit disk graphs. In *Proc. ACM DIALM POMC Workshop*, Sept. 2003.

[19] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: accurate and scalable simulation of entire tinyOS applications. In *Proc. ACM Sensys*, pages 126–137. ACM Press, 2003.

[20] X. Li, Y. J. Kim, R. Govindan, and W. Hong. Multidimensional range queries in sensor networks. In *Proc. ACM Sensys*, Los Angeles, CA, USA, Nov. 2003.

[21] J. Newsome and D. Song. GEM: Graph embedding for routing and data-centric stroage in sensor networks with geographic information. In *Proc. ACM Sensys*, Nov. 2003.

[22] A. Rao, S. Ratnasamy, S. Shenker, and I. Stoica. Geographic routing without location information. In *Proc. ACM/IEEE MobiCom*, pages 96–108, Oct. 2003.

[23] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. GHT: A geographic hash table for data-centric storage. In *Proc. ACM WSNA Workshop*, pages 78–87, Atlanta, Georgia, USA, Sept. 2002. ACM.

[24] K. Seada, A. Helmy, and R. Govindan. Localization errors on geographic face routing in sensor networks. In *Proc. IEEE IPSN Workshop*, Berkeley, CA, USA, Apr. 2004.

[25] R. Sokal and D. Matula. Properties of Gabriel graphs relevant to geographic variation research and the clustering of points in the plane. *Geographical Analysis*, 12:205–222, 1980.

[26] G. Toussaint. The relative neighborhood graph of a finite planar set. *Pattern Recognition*, 12(4):261–268, 1980.

[27] J. Zhao and R. Govindan. Understanding packet delivery performance in dense wireless sensor networks. In *Proc. ACM Sensys*, Los Angeles, CA, November 2003.

Sustaining Cooperation in Multi-Hop Wireless Networks

Ratul Mahajan Maya Rodrig David Wetherall John Zahorjan

University of Washington

Abstract – Multi-hop wireless networks are vulnerable to free-riders because they require nodes to forward packets for each other. Deployed routing protocols ignore this issue while proposed solutions incorporate complicated mechanisms with the intent of making free-riding impossible. We present Catch, a protocol that falls between these extremes. It achieves nearly the low mechanism requirements of the former while imposing nearly as effective barriers to free-riding as the latter. Catch is made possible by novel techniques based on anonymous messages. These techniques enable cooperative nodes to detect nearby free-riders and disconnect them from the rest of the network. Catch has low overhead and is broadly applicable across routing protocols and traffic workloads. We evaluate it on an 802.11 wireless testbed as well as through simulation.

1 Introduction

Selfish behavior is an important design consideration whenever parties with varied interests come together to achieve a common goal. Examples where individual behavior can be at odds with the system goal include free-riding in peer-to-peer file sharing networks [1, 36, 25, 32, 13, 41], cheating in online games [33, 6], ISP competition in Internet routing [38, 12], and network congestion control [16, 17, 37, 23, 4, 26, 28]. As has been observed in many of these systems, some parties will behave selfishly if there is gain to be had, even to the detriment of others.[1] A high-level goal in these systems is to design protocols that ensure the system will work well despite selfish behavior.

In this paper, we study the problem of selfish behavior in multi-hop wireless networks. The emergence of these networks is being driven by the rapid deployment of 802.11 networks and the advantages of relaying packets between nodes. In infrastructure rich areas, relaying can reduce dead spots, lower power consumption [31], and increase network capacity [19]. In rural or developing areas, multi-hop wireless networks can be deployed more readily and at lower expense than traditional wireless networks. Research examples of multi-hop networks include MIT's Roofnet [35], Microsoft's MUP [2], the Digital Gangetic Plains Project [8], and UCAN [27].

Selfish behavior is a concern in this setting because relaying packets for others consumes bandwidth and energy. Unlike traditional, wired LANs, nodes in these networks are often controlled by independent and potentially competing parties, e.g., nearby apartments [2, 35] or villages [8]. In the absence of any pressure to behave cooperatively, nodes have an incentive to *free-ride* by sending their own packets without relaying packets for others. This concentrates traffic through the cooperative nodes, which decreases both individual and system throughput, and may even partition an otherwise connected network.

Deployed routing protocols ignore the issue of free-riding. They simply assume that factors external to the routing protocol cause all nodes to cooperate. This incurs no overhead but unfortunately makes it trivial for a node to free-ride, e.g., by using a simple firewall rule to render itself indistinguishable from a node that lacks the wireless connectivity needed to relay traffic. Moreover, we show experimentally (Section 5.2) that free-riders can obtain substantial benefits. We should reasonably expect free-riding to become prevalent in all but the most benign situations.

Proposed solutions typically incorporate enough mechanism in the routing protocol to eliminate free-riding. This often involves some form of distributed accounting that allows each node to consume no more forwarding service than it provides. These solutions suffer from two serious drawbacks. They require infrastructure that seems unlikely to come about in practice, e.g., centralized clearance services [44, 34] or trusted hardware [11]. And they impose overly restrictive requirements on the system, e.g., uniform traffic rates among all node pairs [39].

Our goal is to combine the strengths of these two approaches while avoiding their weaknesses. Like deployed protocols, we assume that most (but not all) nodes will behave cooperatively. Like proposed solutions, we do not rely on trust alone but include mechanisms that actively discourage free-riding. The insight underlying this combination is that early users of a system are typically cooperative (as they try to get the system to work at all) while selfish behavior emerges when the user base grows [22]. Evolutionary game theory predicts that free-riding will not flourish if discouraged from an early stage [18].

Our solution is called Catch. It uses an existing majority of cooperative nodes to collectively discourage a minority of selfish nodes from free-riding. In game theory parlance, Catch assures that cooperation is an evolutionarily stable strategy. To achieve this, Catch uses novel techniques based on anonymous messages (in which the identity of the sender is hidden) to tackle two critical problems. First, Catch allows a cooperative node to determine whether its neighbors are free-riding, i.e., dropping packets that should be relayed. Second, it enables the cooperative neighbors of a free-rider to disconnect it from the rest of the network. These tasks can be accomplished even when cooperative nodes can communicate with each other only through potential free-riders. The result is that free-riding that previously succeeded is now deterred in a low-cost manner.

We have implemented and evaluated Catch on an in-building 802.11b testbed. This provides a realistic evaluation environment with the complex link quality factors that affect actual wireless systems. Real wireless conditions significantly complicate the implementation of robust mechanisms where nodes monitor the behavior of their neighbors. Yet they have received little attention in earlier work, which to our knowledge is exclusively based on simulation. We find that Catch is able to detect free-riding by individual nodes both quickly and with high accuracy. Its overhead is modest, roughly 24Kbps of control packets per node in our testbed, with no space overhead or cryptographic operations per data packet.

The rest of this paper is organized as follows. We describe our problem setting in Section 2, followed by our approach based on anonymous messages in Section 3. The Catch protocol itself is described in Section 4. Section 5 describes our evaluation based on the 802.11 testbed. We then report simulation results that analyze Catch across a broad range of parameters in Section 6. Finally, we present related work in Section 7 and our conclusions in Section 8.

2 Problem

We focus on selfish behavior, whereby a node gains at the expense of others, rather than malicious behavior, in which a node actively attacks others, e.g., by jamming its radio transmissions. Consider the simple example of a multi-hop wireless network in Figure 1. Here A may wish to send a message to C, either to communicate with C itself or because C serves as a gateway to additional nodes. Because A and C are not in each other's radio range, communication between then must rely on B. On the other hand, B may be interested in communicating via C but uninterested in obtaining any service from A. In that case, B may want to avoid the costs of forwarding packets for A.

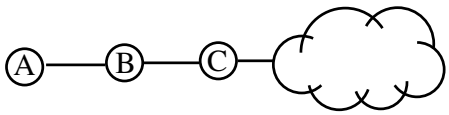


Figure 1: *An example multi-hop wireless network topology in which free-riding can take place.*

B can avoid these forwarding loads in two distinct ways: at the forwarding level and at the routing level. At the forwarding level, B can simply drop some or all of the data packets it receives for forwarding from A. At the routing level, B can refuse to send routing messages that acknowledge connectivity with A. Consequently, B will appear to be a "dead-end" from C's perspective and unreachable from A's, and so neither will ever request forwarding of it. This strategy, which we call *link concealment*, is broadly applicable and, to our knowledge, no existing wireless routing protocol or policing scheme counters it. Our protocol, Catch, prevents B from getting away with these selfish behaviors in the case that both A and C behave cooperatively. B would appear to be immune from adverse consequences for free-riding, because at best only A is aware of either of these behaviors (and it cannot communicate with C except through B), and only C can inflict any punishment on B. But we will see that this is not so.

Catch relies on three assumptions about nodes. First, most of them are cooperative in that they correctly run a protocol we define. A minority of nodes may be selfish and attempt to free-ride; we do not consider collusion amongst these nodes. Second, we assume omni-directional radio transmitters and antennas, so that nodes can overhear nearby communications. This is true for common 802.11 hardware today. Third, nodes have an unforgeable identity. Such identities are not provided by current hardware but can be implemented by other means, e.g., using one-way hash chains [20] and imposing a startup cost for new identities.

Catch does not make any assumption regarding the routing protocol, traffic workload, or objectives of the nodes (such as bandwidth maximization or energy conservation). We believe that it works largely unchanged across these variables. We do not directly consider fairness issues but assume that a higher layer protocol decides what fraction of packets a node should relay for others. Catch can then be used to enforce that policy.

3 The Power of Anonymity

At a high-level, our approach is to use cooperative nodes to monitor for the presence of free-riders and to isolate them from the rest of the network. In this way, free-riding is no longer attractive. However, this approach requires us to tackle two problems, each of which is difficult or impossible to solve in the general case:

1. A node must be able to distinguish between selfish nodes that deliberately drop packets and cooperative nodes that simply do not receive them due to wireless transmission errors. It must do this from afar, even though packet reception events are not externally observable.

2. When a node detects a free-rider, it must be able to signal all of the free-rider's neighbors so that they can collectively isolate it. This must happen even when the only path to those neighbors is through the free-rider itself (which can simply refuse to forward messages that are not in its interest).

We show that anonymous messages, in which the receiver cannot determine the identity of the sender, can be combined with the broadcast nature of wireless to address both problems. This building block was first used in *Cocaine* [40]. Anonymous messages can be provided for most current 802.11 hardware by scrubbing the source MAC address on packets [7]. This forces would-be free-riders to engage in sophisticated games with signal strength measurements if they are to infer the sender. For now, we assume that anonymity can be provided and return to the impact of signal strength hints in Section 5.

3.1 Anonymous Challenges and Watchdogs

To distinguish deliberate packet dropping from wireless errors, we compare an estimate of the true connectivity of a node with its observed forwarding behavior. We use a watchdog [29] to observe the forwarding behavior of a *testee* node that is being tested for selfish behavior from a *tester* node that is assumed to operate correctly. (We use the terms *tester* and *testee* in these roles throughout this paper.) The watchdog relies on the broadcast nature of wireless transmissions. After a node sends a packet to a neighbor for relaying, it can listen to the wireless medium to observe whether the packet is forwarded by the neighbor. It can thereby build up an estimate of the neighbor's forwarding behavior over time.

It is more difficult to remotely estimate the true connectivity of a node. To do so, we develop an *anonymous challenge message* (ACM) sub-protocol as follows. Observe that even a selfish testee must depend on at least one of its testers to forward its packets if it is to stay connected. Call this tester the gateway. Let the gateway regularly but unpredictably send an anonymous challenge to the testee for it to rebroadcast; the gateway refuses to forward packets for the testee if it does not overhear the rebroadcasts (since it believes the testee is not connected or is free-riding). Now consider that all other testers with connectivity to the testee are also sending it anonymous challenges, requiring that they be rebroadcast. Because the testee cannot differentiate gateway challenges from other challenges, it must rebroadcast them all or risk losing connectivity to the gateway. This allows the other testers to estimate their connectivity to the testee. They then compare this to the observed forwarding behavior and infer deliberate packet dropping if there is a discrepancy. In practice, the estimates of connectivity and forwarding are statistical and only recent estimates are compared to allow for real wireless losses.

The ACM protocol is difficult to undermine even with weak anonymity because the likelihood of correctly handling a series of challenges decreases exponentially over time. Without breaking the protocol, a testee has only two options to avoid being flagged as deliberately dropping packets. First, it can be honest and reveal its true connectivity to its neighbors and forward their packets. This is what we desire. Second, it can selfishly drop both challenges and data packets in equal amounts and appear to be poorly connected to *all* its neighbors. But this is a counter-productive strategy. Because the challenges are anonymous they will be dropped independently of their source, and so data packets must also be dropped independently of their source to match. This forces the selfish node to drop and retransmit even its own packets, needlessly consuming its own resources. We note that the ACM protocol is compatible with nodes that sleep for power management, effectively dropping all packets. These nodes neither contribute to the network nor consume its resources, which we consider acceptable behavior. The ACM protocol also has the effect of discarding asymmetric links as does the 802.11 MAC.

3.2 Anonymous Neighbor Verification

Once a tester detects free-riding, it informs all other testers of the free-rider, so that they can simultaneously isolate it. This is necessary: if testers independently break connectivity with the free-rider, they only help the free-rider by reducing its forwarding burden while leaving it able to send its own packets through other testers. The challenge is to inform the other testers even though the only path to them might be via the free-rider, who may discard any incriminating information.

We define an *anonymous neighbor verification* (ANV) sub-protocol to allow a tester to reliably inform the other testers when the testee misbehaves. It operates in two phases. In the first ("ANV Open") phase, all testers become aware of each other via the testee: each tester sends a cryptographic hash of a randomly generated token to the testee for it to rebroadcast, and other testers take note when the rebroadcast happens. As before, anonymous messages are used to prevent the testee from selectively excluding testers. If the testee does not rebroadcast these messages, the testers assume that it lacks connectivity or is free-riding and do not relay packets for it.

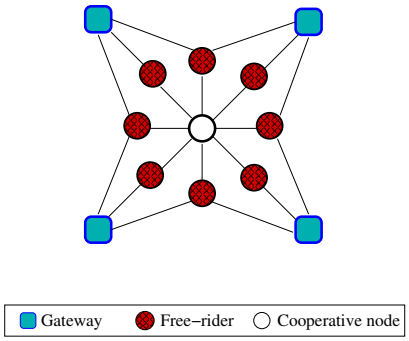


Figure 2: *An example topology to illustrate the use of* Catch. *The lines connect nodes that can directly communicate. (This is done to simplify the illustration; in reality, wireless connectivity is not binary but varies over a range [3].)*

In the second ("ANV Close") phase, each tester releases its token to the testee only if the testee has behaved well, as determined by the ACM protocol. The testee rebroadcasts this token. If the hash of the received token matches one of the hashes collected during the first phase, other testers know that this particular tester is satisfied; the original token can only be released by the tester who encrypted it because it is computationally hard to invert the hash. If a tester does not eventually hear all of the tokens it expects based on the first phase, it concludes that another tester is signaling the presence of a free-rider by refusing to release its token. The free-rider is then isolated by all testers. Note that it is crucial that failure of the testee be signaled by the *absence* of a message to prevent the free-rider from blocking the signal, as it could with a more straightforward positive signaling mechanism.

We make two further observations. First, as before, dropping messages in the first phase to exclude particular testers and their data packets is unlikely to succeed. This is because the likelihood of correctly matching anonymous messages to testers decreases exponentially over time. Second, interference in the second phase of the sub-protocol by the testee is clearly unproductive because it can only lead to its isolation.

3.3 Example

We use an example to illustrate the power of the combined protocols. In Figure 2, a cooperative client is completely surrounded by free-riders. Without Catch the client cannot communicate with any of the gateways because the free-riders ignore its packets. With Catch, the client uses the ACM protocol to determine that it is in fact connected to the selfish nodes, and the watchdog to verify that its packets are not being dropped. If they are, the client uses the ANV protocol to inform the gateways, which isolates the free-riders. The threat of this punishment deters free-riding. Further, while we leave the issue of collusion for future work, Catch works for this topology even if the selfish nodes collude. This suggests a degree of collusion-resistance in the design.

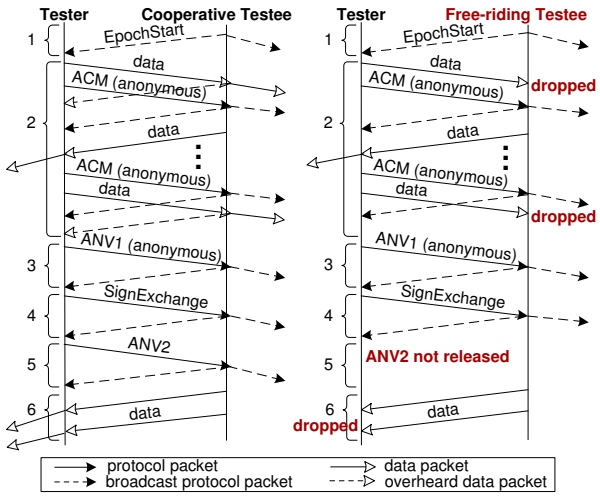


Figure 3: *Protocol flow. Packet exchange between a tester and a cooperative (left side) or free-riding (right side) testee. Numbers on the left of the time sequence correspond to the protocol steps.*

4 The Catch Protocol

Catch builds on the anonymous techniques above, adapting them for use in real, wireless networks.

4.1 Overview

Catch operates as a sequence of protocol epochs run between a *testee* node and its neighbors, who act as *testers*. Figure 3 provides two illustrations of the per-epoch protocol steps, one when the testee is cooperating and the other when it is free-riding.

Each epoch consists of the following steps:

1. *Epoch-Start.* The testee broadcasts an EpochStart packet that includes its identity and an epoch identifier. Nodes that receive this request participate as testers for this epoch.
2. *Packet Forwarding and Accounting.* Testers run a watchdog [29] to count the number of their data packets that were correctly relayed. Note that the watchdog allows the testers to check for packet reordering (to force TCP backoff), corruption, or misdirection. Simultaneously, testers run the ACM protocol to estimate true connectivity. This involves sending anonymous challenges and counting their rebroadcasts; the data packets themselves are not anonymous.

3. *Anonymous Neighbor Verification Open (ANV1).* Each tester "opens" the two-phase ANV subprotocol (Section 3.2) by sending an anonymous packet containing a nonce (to prevent replay attacks) and a hashed token to the testee for rebroadcasting.

4. *Tester Information Exchange.* Each tester compares the fraction of its data packets that it overheard and the fraction of its anonymous challenges that it heard reflected. It obtains a one-bit ("sign") result depending on which is greater: 0 for challenges and 1 for data packets. It then sends its sign bit and identity to the testee for rebroadcasting.

5. *Epoch Evaluation and ANV Close (ANV2).* Each tester determines whether the testee is operating correctly using its observations and the sign bits from other testers. This is done with a pair of statistical tests described in the next subsection. If both tests pass (and the testee correctly rebroadcast the tester's sign bit), the tester releases its token. Otherwise, it withholds its token.

6. *Isolation Decision.* An epoch fails for a tester if it withholds its token or it does not receive all expected tokens. If too many epochs fail too quickly (Section 4.3) then the tester decides that the testee is free-riding and punishes it by dropping its packets for a fixed number of epochs. By virtue of the protocol, all testers decide to punish a free-rider at (nearly) the same time, so that it is isolated.

We increase the likelihood of all testers seeing all control packets in two ways. First, we use retransmissions if a tester does not hear the rebroadcast. Second, we use cumulative broadcasts, where the testee sends all of the information it has received on every transmission.

4.2 The Per-Epoch Tests

Each tester applies two statistical tests per epoch to determine whether a testee is behaving correctly. Each test is designed to be sensitive to distinct selfish strategies. The key challenge in both is to avoid mistaking volatile wireless conditions for misbehavior.

One selfish strategy is to drop packets from a particular tester in the hope that the consensus across neighbors will be that the free-rider has passed the epoch, since all other testers should find its behavior acceptable. To detect this, each tester compares observed forwarding and true connectivity estimates for the last three epochs using the z test [30]. We found that high confidence levels (99% and above) coupled with using measurements from multiple epochs provides a good balance between quick detection of free-riding and a low rate of false positives.

The second selfish strategy is to uniformly drop some fraction of the packets received from each tester, making it hard for any one of them to conclude that free-riding has taken place. To detect this, we employ the sign test [30] using the sign bits exchanged by all testers. This test is based on the idea that the perceived forwarding and connectivity rates should have identical means if the testee is not deliberately dropping packets. Thus, random fluctuations in each epoch should yield about as many results in which one exceeds the other as the opposite. Each tester accumulates the one-bit results for all epochs in which it has participated, and applies the sign test to decide if the balance is reasonable.

4.3 The Isolation Decision

Isolation of a testee is decided by all testers in parallel. Each maintains a small history of per-epoch test results, represented as a three state finite state automaton (FSA) that moves to the right when an epoch fails and the left when an epoch passes. If the FSA falls off the right edge, the testee is isolated.

While it might seem that this scheme allows a node to free-ride for at least half of the epochs, the fact that the per-epoch test results depend on packet accounting data aggregated over the previous three epochs prevents this: free-riding in any one epoch impacts the tests for three consecutive epochs, and is likely to lead to multiple failed tests. We more fully explore this issue in Section 6.3.

4.4 Protocol Fail-safes

Because Catch is designed to operate when some nodes act in a selfish manner, we are as concerned about what happens when the protocol is not followed as when it is. In Appendix A we provide a short analysis by message type that shows that selfish nodes cannot undermine the protocol in the absence of collusion.

5 Experimental Evaluation

This section describes our experiments with Catch on an 802.11b testbed. This allows us to test how well Catch works in wireless environments that exhibit complex packet loss behaviors [24].

5.1 The Testbed

Our testbed is composed of 15 PCs equipped with 802.11b that run Linux 2.4.26. We use NetGear MA311 PCI network adapters (Prism 2.5 chipset), operating in the ad-hoc mode on channel 1 using the *hostap* driver. Each node also has a wired Ethernet interface to facilitate remote management of the experiments.

The testbed is located on a single floor of an office building, as shown in Figure 4. The building has its own

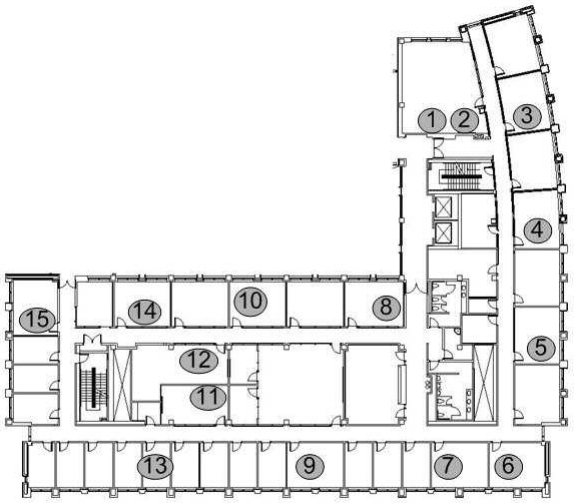


Figure 4: *Our wireless testbed, consisting of fifteen 802.11b nodes. The node locations are marked with circles. Horizontally, the building is 184 ft. long.*

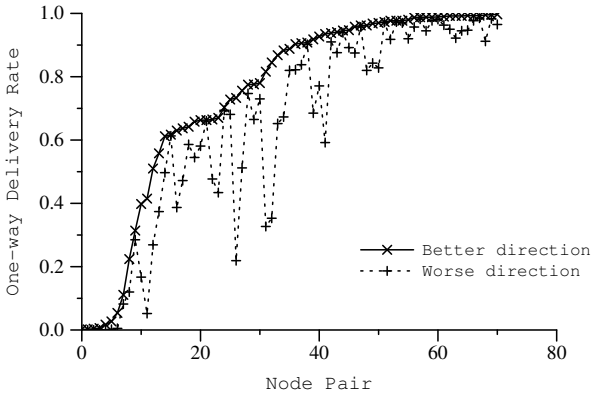


Figure 5: *For each node pair in the testbed, the fraction of sent packets successfully received in each direction. There are 105 pairs total in the testbed. Only node pairs with a non-zero delivery rate between them in at least one direction are shown.*

dense deployment of wireless access points, including ten on the same floor as our testbed, some of which compete with us on channel 1. Such a setting is noisy, but realistic [3].

Our system exhibits well-known characteristics of wireless networks, including error rates that are not a simple function of distance, that are strongly asymmetric, and that vary widely over time. Figure 5 gives a static summary of these effects. It shows the average one-way delivery rate in each direction for each pair of nodes that were able to communicate at all. To compute these rates, each node broadcast 500 1000-byte packets over two minutes. The other nodes counted how many of those packets they received. The figure shows a wide range of delivery rates rather than a binary state of connectedness, which is consistent with prior results [3, 43]. The diameter of our network is between 3 and 5 hops, depending on the threshold of link quality at which two nodes are considered connected.

5.1.1 Catch Implementation

We implemented Catch at user-level using the Linux *netfilter* framework to monitor and manipulate the packets sent, received, and forwarded by a node. The watchdog component of Catch also needs to overhear all packets sent by the node's neighbors regardless of their intended destination. To capture these packets, we operate our wireless network adapters in promiscuous mode and use the Linux pcap framework. The Catch protocol itself is written in ruby and is completely independent of the underlying routing protocol.

One complication is that the watchdog mechanism needs to account for 802.11 MAC-level retransmissions. To see this, consider a tester judging whether the testee forwarded a particular data packet. The quality of the link between the testee and the recipient determines the number of retransmissions done by a cooperative testee. This in turn changes the probability that the tester will overhear the transmission. To correct for this recipient-based variation, we measure the data forwarding rate using only the first transmission as indicated by a bit in the 802.11 MAC header. A complete implementation of Catch would also check that retransmissions are handled consistently to close a secondary loophole. We have not done so yet.

We use the following parameters values for our experiments; simulations suggest that Catch is not highly sensitive to the exact choices. The length of an epoch is set to one minute. The confidence interval for the z test is 99.999%, and that for the sign test is 99.995%. (Both experiments and simple analysis showed that very high confidence values are most effective.) There are fifteen anonymous ACM messages per epoch, each of which is 1500 bytes, the MTU (maximum transmission unit) size of our network adapters. The loss rate for smaller data packets (such as TCP acknowledgements) can be less than that of the ACM messages. To verify forwarding behavior, our implementation checks that the loss rate for data packets is less than that for ACM messages.

5.1.2 Multi-Hop Performance

We first show the potential benefit of relaying packets by comparing the performance of a single, centrally located access point (AP) setup to that of multi-hop routes. To do this we transfer a large file from one node, which acts as the AP (node 8 in Figure 4), to four client nodes (nodes 4, 6, 9 and 14). Each client downloads a 600KB file ten

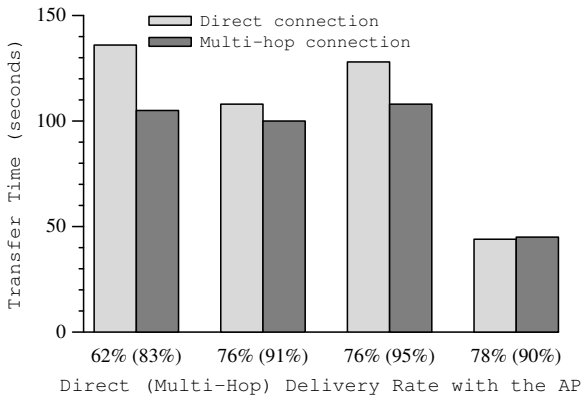


Figure 6: *Time to transfer 6MB from node 8 in Figure 4 by four other nodes via direct and multi-hop connections. The x-axis label gives the delivery rate of the direct connection with node 8, with the delivery rate for the multi-hop path in parentheses. The client nodes, from left to right, are 4, 6, 14, and 9.*

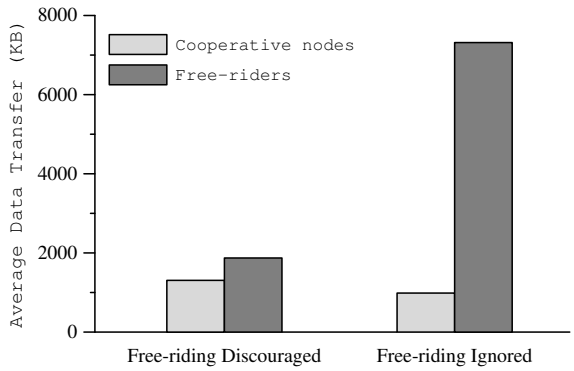


Figure 7: *Average amount of data transferred per node when free-riding is discouraged and when it is not. None of the nodes free-ride in the former. Nodes 7, 14 and 15 free-ride in the latter.*

times. In one set of experiments, the clients communicate directly with the AP. In the other, they use multi-hop routes via a single intermediary node, over paths 4:5:8, 6:7:8, 14:10:8, and 9:10:8. We use static routes between nodes to factor out effects that stem mainly from the routing protocols; wireless routing protocols are an open area of research [14].

Figure 6 shows the results. The x-axis labels give the delivery rate of the direct links, averaged over both directions. The parenthesized numbers give an estimate of the quality of the two-hop path, computed as the product of the delivery rate of the individual links. In total, the use of multi-hop paths reduced download time by 16%, with per-node benefits ranging from 30% to -2%. The better performance of the multi-hop routes is due in part to the lower packet loss rates they enjoy. De Couto *et al.* have studied these issues in more detail [15, 14].

5.2 The Impact of Free-riders

We now consider the performance impact of free-riding, both as benefits to the free-riders and as costs to the cooperative nodes. We do this by contrasting the per-node throughput achieved in a fully cooperative network with those achieved when some nodes are allowed to free-ride.

In this experiment, we randomly selected 3 nodes as free-riders. All nodes were trying to download randomly selected files from randomly selected servers. Figure 7 illustrates the average amount of data transferred under the two scenarios: "Free-riding Discouraged," which results in all nodes behaving cooperatively, and "Free-riding Ignored," where free-riders simply do not relay packets for cooperative nodes. Both scenarios were run for 35 minutes. The two bars in each scenario average the per-node results for twelve nodes that acted cooperatively and for the three free-riders. The data illustrates two key points. First, there is a very large incentive to free-ride: the free-riders improve their throughput by 400% relative to when they are forced to cooperate. This indicates that there is considerable potential motivation for nodes to behave selfishly in these environments if they can do so without retribution. Second, the improved situation for the free-riders comes at the expense of cooperative nodes. The performance of the cooperative nodes is decreased by 25% when 20% of their fellow nodes selfishly misbehave. While this is only a single example, it clearly demonstrates the need to incorporate protection against free-riding in routing protocols.

5.3 Catch Evaluation

In this section we evaluate the effectiveness of Catch.

5.3.1 Detecting Free-riders

Our first experiment measures the speed with which Catch detects free-riding. To construct a base case, we selected triplets of nodes such that both the first and the third node had a reasonable (>75%) delivery rate to the second node. The second node was configured to act as a free-rider that randomly dropped a fraction of the packets it received for forwarding. We experimented with different drop rates; Drop rates less than 100% mimic a situation in which the free-rider tries to evade detection by appearing to be a cooperative but poorly connected node. The first node downloaded randomly selected files ranging from 1KB to 3MB in size from the third node. The request and response traffic was relayed through the second node. Five download sessions ran in parallel so that even in the presence of a high drop rate and TCP backoff dynamics, a minimum amount of traffic (roughly ten packets per epoch) is generated for the statistical tests.

Figure 8 presents the results. The line "Drop packets from both" corresponds to the case when the free-rider drops packets from both neighbors. It shows the average number of epochs required to detect a free-rider for

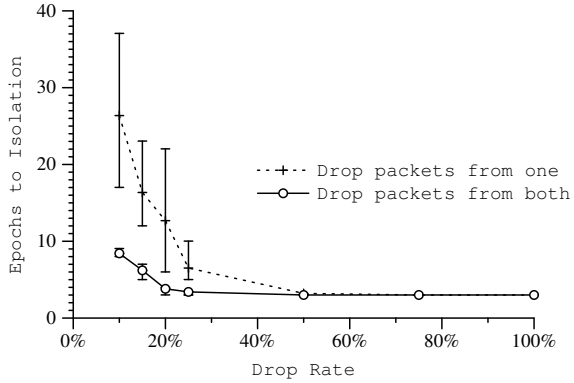


Figure 8: *The number of epochs required to detect free-riders in the testbed versus the fraction of packets a free-rider dropped. Each point is the average of 10 experiments. Vertical bars represent the inter-quartile range.*

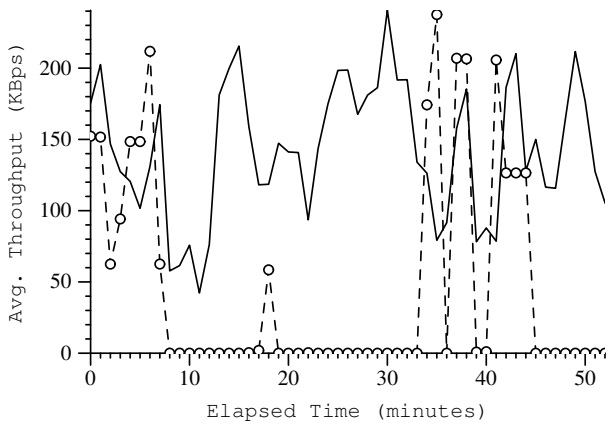


Figure 9: *Average throughput of cooperative nodes (solid line) and free-riders (dashed line) as a function of time. Throughput was calculated using one minute intervals. There were three free-riders. The punishment interval is 30 minutes.*

varying drop rates. Catch reacts quickly to free-riding, and its reaction time decreases with drop rate. Detection is almost immediate for very high drop rates; recall from Section 4.3 that at least three epochs must fail before isolation. Even at the low drop rate of 10%, Catch isolates the free-rider in under 9 epochs.

The curve "Drop packets from one" shows the results for the case where the free-rider dropped packets only for the client. This evaluates whether a single victim can cause the free-rider to be isolated. We find that for high drop rates the detection speed is just as fast as the previous case. It is slower at lower drop rates, but even at the low drop rate of 10% the average detection time is less than 30 epochs. Thus, a free-rider that persistently drops packets of just one neighbor at a very low rate is eventually caught and punished.

5.3.2 False Accusations

We next check that the rapid detection of free-riders does not come at the cost of falsely accusing cooperative nodes of free-riding. We ran two five hour experiments in which all nodes were cooperative. Each node repeatedly downloaded files (as before) from randomly chosen servers. This workload is high enough to saturate our network, stressing the accuracy of inference and increasing the probability of false accusations. We observed no false positives in the first experiment and a single false positive in the second. It is difficult to estimate the true rate of false accusations from this because they are so rare, but nevertheless we find it encouraging.

5.3.3 Coordinated Isolation

We now evaluate whether wireless conditions hinder the ability of the testers to simultaneously isolate a free-rider.

We randomly selected three (20%) nodes as free-riders that dropped all the packets they received for forwarding. All nodes executed a workload similar to the one in the previous section with the exception that nodes only selected the cooperative nodes as file servers. We then measured the throughput obtained by the free-riders. It should be zero if coordinated isolation was successful.

Figure 9 plots the average throughput obtained by the free-riding and cooperative nodes. It shows that the cooperative nodes successfully shut out the free-riders. Roughly eight minutes into the experiment, all the free-riders were identified and isolated. Though not shown in the graph, the spread of time over which different neighbors of a free-rider started isolating it was two minutes. The free-riders were allowed to send traffic again after the punishment interval of 30 minutes. The average throughput of the free-riders appears to recover before 30 minutes because different free-riders were isolated and released at different times.

5.3.4 Protocol Overhead

We report on the overhead of Catch in this section. We have made no attempt to optimize the protocol because its requirements are already modest.

Consider the activity for a pair of neighboring nodes in an epoch, both playing the role of tester and testee. The packet overhead of Catch comes from its messages, which have different sizes and frequencies: StartEpoch (40 bytes), ACM challenges and responses (1500 bytes, 15 times per epoch), ANV open and close (100 bytes), and sign exchanges (40 bytes). These packets come to a total of 0.6 packets or 758 bytes per neighbor per second. Our testbed has fewer than four well-connected neighbors per node on average, which means that the protocol overhead is less than 2.4 packets per second or 3 KBps per node. This is 3% of the 100 KBps that the honest nodes got on average in Figure 9. The overhead would be even lower for the newer and faster 802.11a/g.

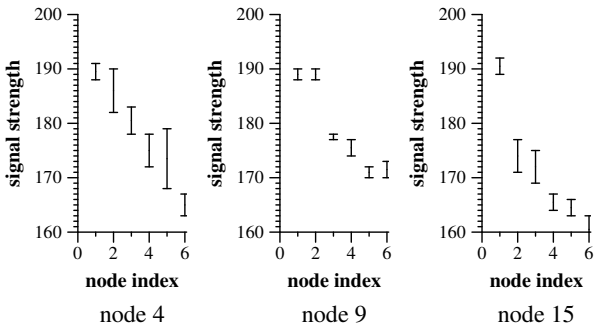


Figure 10: *The spread of received signal strength at Nodes 4, 9 and 15 in our testbed. The y-axis represents the magnitude of the signal reported by the hardware. The bars represent the range in which 90% of the packets from a neighboring node fall.*

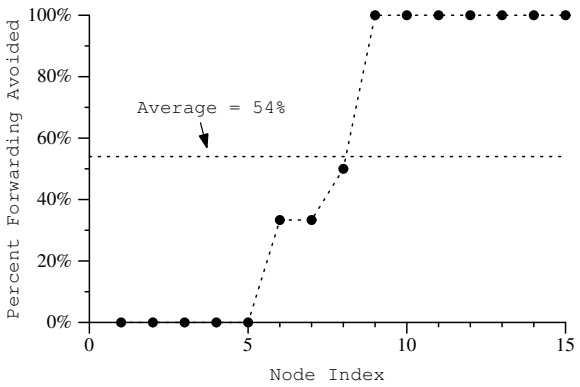


Figure 11: *The fraction of forwarding load avoided if a node adopts a signal strength cheating strategy. (We assume forwarding load is proportional to the number of neighbors.)*

We found the processor consumption of Catch to also be very reasonable. Informally observed using *top* during our experiments, it took at most 10% of the CPU on Pentium-IV 3 GHz nodes. Much of this is an artifact of our user-level implementation. Each packet that passes through the local machine or is promiscuously overheard crosses the user-kernel boundary at least once. In fact, before moving to a PC-based testbed for OS reliability reasons, we had successfully experimented with Catch on a testbed composed of 10 iPAQs.

5.3.5 Compromising Anonymity

In this section we study the potential leverage of signal strength attacks on anonymity. We show that even in its present form Catch is useful in protecting the cooperative nodes and is by far preferable to doing nothing. Taking specific steps in Catch to discourage signal strength based cheats is the subject of future work.

At the MAC level, anonymity is a reasonable assumption, since it is possible to send packets with an arbitrary source address and contents using commonly available 802.11 hardware [7]. At the physical level, however, strong anonymity cannot be guaranteed against a determined adversary: the source of a packet might be estimated, or at least classified, from the wireless signal strength or direction.

Signal strength cheats are a level of escalation beyond the selfish misbehavior we have defended against thus far. Free-riding using signal strength measurements is not a simple matter of installing a firewall rule, but requires changes to the network interface driver. Our hardware cannot give information about signal source direction, nor can any commodity hardware (fitted with an omnidirectional antenna) of which we are aware.

Catch provides protection against such cheats because the received signal strength from an individual neighbor varies over a range of values. When the ranges of multiple neighbors overlap it becomes impossible to accurately distinguish among them. Empirical reports of wireless network conditions [42, 43, 24] and localization schemes based on received signal strength [5] illustrate the difficulties of using signal strengths. As examples, Figure 10 shows the spread of received signal strength at three nodes in our testbed.

To better understand the overall threat, we experimented with a cheater that uses signal strength to differentiate among its neighbors. The cheater listens to data packets for a short period of time, measuring their signal strengths and sources. It then chooses a signal strength threshold at which to drop incoming packets. It relays packets and appears cooperative to neighbors whose packets arrive with strengths above the threshold. It drops packets below the threshold to appear to be a legitimate non-neighbor to all other nodes.[2] Using this procedure, a cheater may end up cooperating with between just one and all of its legitimate neighbors. Of the nodes in Figure 10, Node 4 is forced to cooperate with all of its neighbors, Node 9 with only two of them, and Node 15 with only one of them. (Peripheral nodes that can uniquely identify a neighbor do not present a major threat as such nodes are not expected to relay packets.)

Figure 11 shows the benefits of this attack in our testbed. For each of the 15 nodes we plot the fraction of forwarding traffic that would be avoided, assuming that forwarding loads are proportional to the number of neighbors, or zero if a cheater manages to establish only a single neighbor. We conservatively assume that when a cheater identifies a subset of its neighbors, one of the nodes in the subset is capable of forwarding packets for it; otherwise, the cheater needs to admit connectivity to other neighbor(s). Just under half the time a cheater can escape forwarding entirely, while just over half it avoids none or only a modest amount. Of course, if no protocol is run to protect against cheating, all nodes can cheat 100%, leading to a tragedy of the commons.

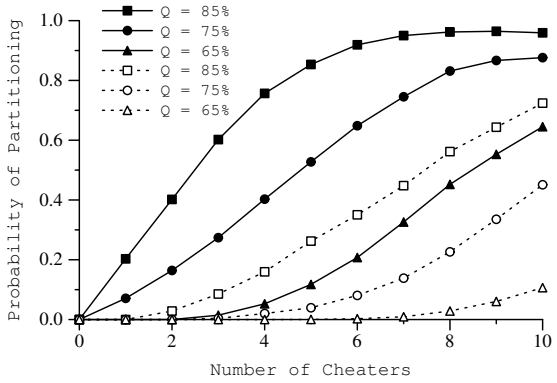


Figure 12: *Probability that the cooperative nodes are partitioned versus varying numbers of (randomly chosen) cheating nodes when running with (dotted lines) and without (solid lines)* Catch*. Under* Catch *the cheaters use a signal strength based cheating strategy. Only links with delivery rates at least Q are considered useful.*

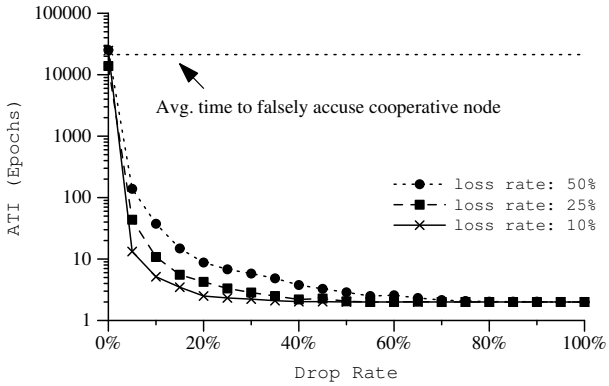


Figure 13: *Average time to isolation versus drop rate, for various background network loss rates.* (Y-axis on a log scale.)

Even though a cheater may expect to reduce its forwarding load by about half using signal strength information, Catch still helps the cooperative nodes. Figure 12 shows that Catch greatly improves connectivity for those nodes, relative to taking no measures against cheating. It plots the probability that a (randomly selected) set of such cheaters would partition the cooperative nodes when running with and without Catch. Because Catch forces many cheaters to admit to multiple neighbors, and so to be available for packet forwarding, it significantly reduces the odds that the network is partitioned. For example, when 20% (3) of the nodes cheat, that probability is lowered from about 60% to about 10% when using the highest quality links. At a 75% link delivery rate threshold, the odds of a network partition are reduced from about 30% to zero. Of course, these results are specific to our testbed; in general, the extent of protection provided by Catch depends on the degree of overlap between the signal strengths of different neighbors. We are currently extending Catch to mitigate such attacks by having testers vary their signal strength as part of the testing.

6 Simulation and Analysis

We now extend our analysis of Catch using simulation.

6.1 Simulation Testbed and Metrics

We built a simulator to generate packet loss and reception counts for each epoch and to drive the protocol state machine. The simulator does not model the details of packet delivery. The protocol state machine is parameterized by the neighborhood topology, its loss rates, and the z and sign statistical test parameters. We focus on packets that are subject to Catch's statistical tests and ignore other (control) packets. Our base setting includes a single free-rider with six neighbors. The epoch duration in the simulations is one minute. We set the confidence levels for the z and sign tests to 99.999% and 99.995% respectively. Results from the simulator showed that these values achieved the best overall tradeoff between detection speed and false positive rate.

To assess the effectiveness of Catch, we use *Average Time to Isolation* (ATI) as the metric. ATI is measured in units of epochs. An ideal policy would exhibit ATI values of one for nodes that free-ride (at any rate), and infinite ATI values for those that do not.

6.2 Physical Environment Effects

We first evaluate Catch's robustness to two characteristics of the physical environment: packet loss and network density. To model free-riding, we use a straightforward strategy in which the free-rider drops packets randomly with fixed probability. Because the packet losses due to the wireless network are also modeled as a random process, this drop strategy is arguably difficult for our statistical tests to detect.

6.2.1 Packet Loss

We would expect higher wireless loss rates to make it more difficult to detect free-riding. Figure 13 shows ATI results as a function of drop rate for three different background network loss rates. Each data point shows the average of 40 runs. When there is no free-riding (the y-axis), there is a large isolation time – an average of around 26,000 epochs (about 18 days). These times fall steeply as the drop rate grows, to under 10 epochs for drop rates of 10-20%. The results for loss rates in the range of 10%-25% are in line with those observed in our testbed (Figure 8), except that the homogeneous link qualities in the simulation environment result in much longer false accusation times. Thus, the impact of high

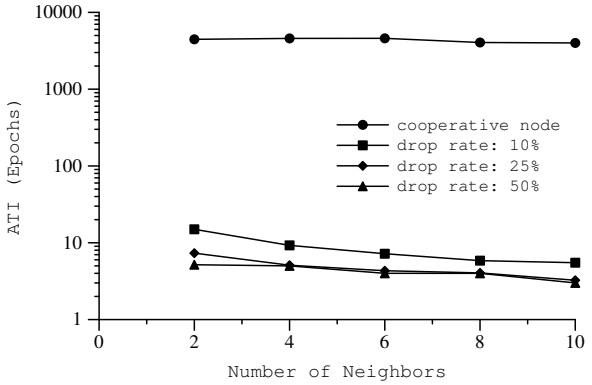


Figure 14: *Average time to isolation versus number of neighbors.* (Y-axis on a log scale.)

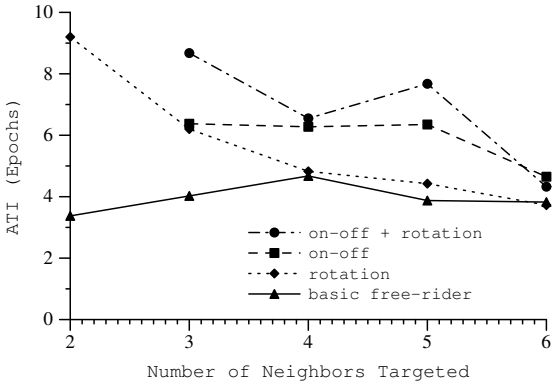


Figure 15: *The time to isolation for customized free-riding strategies. The free-rider directs all misbehavior in a single epoch to the number of neighbors given on the x-axis.* (Total cheat rate = 20%. Network loss rate = 20%.)

wireless loss rates on Catch is quite small. Even at a network loss rate of 50% Catch isolates a free-rider who drops 25% of the packets it needs to forward in seven epochs on average, which is only four epochs more than the fewest possible.

6.2.2 Network Density

We would expect Catch to perform better in denser networks because larger neighborhoods are more likely to make correct statistical decisions. Figure 14 examines the impact of the number of neighbors on detection and false accusation times. We show results for a cooperative node (the top line) as well as for free-riders at drop rates from 10-50%. Increasing the number of neighbors from six to ten yields a small decrease in the time to detect free-riders, as might be expected: already at 6 neighbors there is little room for improvement. More surprisingly, reducing the number of neighbors by a factor of three, to only two, increases detection time by only a few epochs. Additionally, the rate at which cooperative nodes are falsely accused is essentially unaffected over the entire range. Thus, Catch seems to be robust, working well in both high and low density networks.

6.3 More Sophisticated Cheaters

Thus far, we have analyzed a simple drop model in which the free-rider randomly drops packets it is meant to forward. We now use our knowledge of the statistical tests to construct packet dropping variations that target potential weaknesses. While we cannot prove the negative result that there are no strategies that might be effective against Catch, we can show that these customized strategies yield only very limited success.

One variation is targeted free-riding, in which the free-rider drops packets from a time-varying subset of neighbors, rather than uniformly from all. This stresses the z test in Catch, whereas we know that the basic free-rider is most often detected by the sign test. We call this approach "rotation." A second variation attacks the isolation decision process. Since three consecutive failed epoch tests are required to isolate a node, a free-rider may attempt to escape isolation by dropping packets on, say, alternate epochs. We call this the "on-off" strategy. Finally, both attacks may be used at once.

Figure 15 plots the number of epochs to isolation for these strategies against the number of nodes targeted, for the difficult environment where the loss rate is as large as the drop rate. (Both were set to 20%.) The graph suggests that these custom-built strategies are only very modestly successful. The most effective strategy for the free-rider is to obtain its overall average drop rate of 20% by dropping 60% of the packets from two of its six neighbors, while rotating that pair each epoch. Using that strategy, the free-rider is isolated in nine epochs on average, compared to five epochs for the base free-riding strategy.

As another variation of the basic free-riding model, we experimented with free-riders that drop packets in a deterministic pattern, rather than randomly. The threat here is that the reduction in variance will help free-riders avoid detection. In fact, the opposite happened: Catch was more effective.

6.4 Assessing Effectiveness

To complete this section, we consider how much better it might be possible to do than Catch. This is a difficult question to answer. We address it by comparing Catch to an unrealistically powerful alternative, the *Detection Oracle*, that serves as an informal upper bound on what might be possible by any technique.

The Detection Oracle hears all packet transmissions everywhere in the network, without loss, and so has reliable knowledge of all externally visible events. Addi-

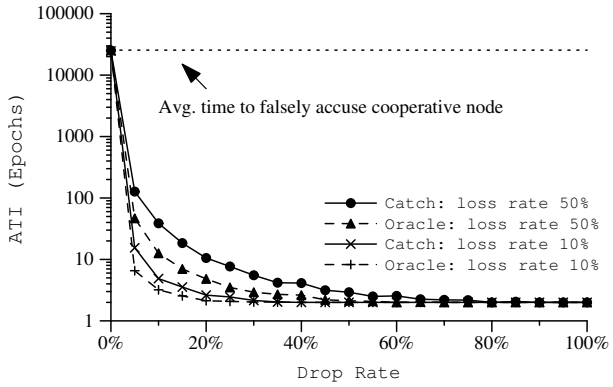


Figure 16: *Comparison of the time to isolation with* Catch *and the Detection Oracle as a function of drop rate for 10% and 50% network loss rates.* (Y-axis on a log scale.)

tionally, it retains infinite history information, enabling it to apply the Catch statistical tests over this maximal pool of data. In contrast, the nodes in any real system have only imprecise information (due to losses), each one is directly aware of only a subset of the global information, and history information must be devalued due to the changing environment.

Figure 16 compares the Detection Oracle with Catch. It suggests that Catch does nearly as well as possible. The oracle's advantage exceeds a five epoch reduction in detection time only in the case of high network loss rate (50%) and relatively low (5-25%) drop rates.

7 Related Work

Anonymous broadcast was first used as a protocol building block in the *Cocaine* protocol for auction between mistrustful parties [40]. In a manner similar to Catch, Cocaine combines this building block with one-way hash functions. We apply this approach in a different and practical setting, and our work also hints at the generality of the building block and the approach.

Catch belongs to the class of enforcement-based mechanisms that discourage free-riding through the fear of punishment. The watchdog part of our detection mechanism was originally proposed by Marti *et al.* [29]. It is our use of it in real networks and in conjunction with anonymity to detect misbehavior that is novel. Existing enforcement-based protocols [29, 10, 9] rely on reputation spreading to deal with cheating nodes. This requires global flooding, while Catch limits information spread to single-hop neighborhoods. Moreover, simple flooding requires network redundancy as selfish nodes will not forward incriminating reputation packets. Catch uses anonymity and one-way hash functions to reliably communicate with the neighbors of free-riders. Our use of one-way hash functions is similar to Hu *et al.*'s work on secure routing in wireless networks [20, 21].

Incentive-based approaches discourage free-riding by making cooperation more attractive. Nodes accumulate virtual currency by forwarding for others, which they can then use for sending their own packets. Examples include Nuglets [11], Sprite [44] and priority forwarding [34]. These schemes rely on a trusted central authority or tamper-proof hardware to ensure the integrity of the currency, and to redistribute wealth so that even nodes that are not in a position to forward for others can send their packets. In contrast, the operation of Catch is completely distributed. Incentives also fail to encourage nodes with very little data of their own to send. This can lead to a disconnected network when light-senders are located at strategic points in the topology.

Finally, game-theoretic approaches formulate the forwarding decision such that forwarding at a certain rate becomes the Nash equilibrium [18] for the network. This means that deviation from the recommended forwarding behavior can only result in situations that are worse for the deviant node. Generous Tit-for-Tat (GTFT) is an example of such an approach [39]. Like GTFT, Catch relies on the mechanics of Tit-for-Tat by assuming cooperation and punishing free-riders. However, while GTFT requires knowledge about the utilities of all the nodes in the network, Catch relies only on information collected in the one-hop neighborhood of individual nodes.

8 Conclusions

We have presented Catch, a protocol to sustain cooperation in multi-hop wireless networks comprised of autonomous nodes. Catch is much more widely applicable than other proposed solutions, needing no central authority and placing no restrictions on workloads, routing protocols or node objectives. It uses novel strategies based on anonymous messages and statistical tests to detect free-riders with high likelihood and punish them with periods of isolation. *Anonymous challenge messages* are used to estimate true loss rates, even when dealing with untrusted and uncooperative nodes. *Anonymous neighbor verification* is used to compel a node to forward packets, even when the data being carried is contrary to its interests. While our application of anonymity and neighborhood watch are specific to the wireless domain, we expect that these techniques are general enough to be applicable in other domains.

We implemented Catch in Linux and performed what to our knowledge is the first evaluation of cooperative routing protocols in an 802.11 wireless testbed. We showed that Catch works well despite volatile wireless conditions and requires little bandwidth overhead (and negligible CPU overhead). In our experiments, free-riders are quickly isolated from the network (and more

rapidly for more egregious drop strategies) and cooperative nodes are rarely accused of misbehaving. Simulations confirm this finding over a wide range of conditions. We quantified the impact of free-riding by showing that the presence of even a few free-riders can partition the network. In one experiment, their presence led to a 25% overall performance degradation for the cooperative nodes. We also explored the leverage of signal strength cheats, and found that even without any measure to actively thwart such cheats, Catch provides worthwhile protection. Extending Catch to defeat these strategies is part of our future work.

9 Acknowledgements

We thank Krishna Gummadi, Ed Lazowska, Hank Levy, Mike Swift, the anonymous reviewers, and Amin Vahdat, our shepherd, for their comments on earlier versions of this paper. Brian Youngstrom provided valuable assistance with setting up the testbed.

This work was supported in part by NSF (Grant No. ANI-0133495), Atheros Communications, and Microsoft Research.

References

[1] E. Adar and B. A. Huberman. Free riding on gnutella. *First Monday*, 5(10), Oct. 2000.

[2] A. Adya, P. Bahl, J. Padhye, A. Wolman, and L. Zhou. A multi-radio unification protocol for IEEE 802.11 wireless networks. Technical Report MSR-TR-2003-41, Microsoft, June 2003.

[3] D. Aguayo, J. Bicket, S. Biswas, G. Judd, and R. Morris. A measurement study of a rooftop 802.11b mesh network. In *ACM SIGCOMM*, Aug. 2004.

[4] A. Akella, R. Karp, C. Papadimitrou, S. Seshan, and S. Shenker. Selfish behavior and stability of the Internet: A game-theoretic analysis of TCP. In *ACM SIGCOMM*, Aug. 2002.

[5] P. Bahl and V. Padmanabhan. RADAR: An in-building RF-based user location and tracking system. In *IEEE INFOCOM*, Mar. 2000.

[6] N. Baughman and B. N. Levine. Cheat-proof playout for centralized and distributed online games. In *IEEE INFOCOM*, Apr. 2001.

[7] J. Bellardo and S. Savage. 802.11 denial-of-service attacks: Real vulnerabilities and practical solutions. In *USENIX Security Symposium*, Aug. 2003.

[8] P. Bhagwat, B. Raman, and D. Sanghi. Turning 802.11 inside-out. In *HotNets-II*, Nov. 2003.

[9] K. A. Bradley, S. Cheung, N. Puketza, B. Mukherjee, and R. A. Olsson. Detecting disruptive routers: A distributed network monitoring approach. In *IEEE Symposium on Security and Privacy*, May 1998.

[10] S. Buchegger and J.-Y. L. Boudec. Performance analysis of the CONFIDANT protocol: Cooperation of nodes — fairness in dynamic ad-hoc networks. In *MobiHOC*, June 2002.

[11] L. Buttyan and J. Hubaux. Stimulating cooperation in self-organizing mobile ad hoc networks. *ACM/Kluwer Mobile Networks and Applications*, 8(5), Oct. 2003.

[12] B.-G. Chun, R. Fonseca, I. Stoica, , and J. Kubiatowicz. Characterizing selfishly constructed overlay routing networks. In *23rd IEEE International Conference on Computer Communications*, Mar. 2004.

[13] B. Cohen. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems*, June 2003.

[14] D. De Couto, D. Aguayo, J. Bicket, and R. Morris. A high-throughput path metric for multi-hop wireless routing. In *MobiCom*, Sept. 2003.

[15] D. De Couto, D. Aguayo, B. A. Chambers, and R. Morris. Performance of multihop wireless networks: Shortest path is not enough. In *HotNets-I*, Oct. 2002.

[16] S. Floyd. Congestion control principles. IETF RFC 2914, Sept. 2000.

[17] S. Floyd and K. Fall. Promoting the use of end-to-end congestion control in the Internet. *IEEE/ACM Transactions on Networking*, 7(4), Aug. 1999.

[18] D. Fudenberg and J. Tirole. *Game Theory*. MIT PRess, Aug. 1991.

[19] P. Gupta and P. R. Kumar. The capacity of wireless networks. *IEEE Transactions on Information Theory*, IT-46, Mar. 2000.

[20] Y.-C. Hu, D. Johnson, and A. Perrig. Secure efficient distance vector routing for mobile wireless ad hoc networks. In *IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, June 2002.

[21] Y.-C. Hu, A. Perrig, and D. Johnson. Ariadne: A secure on-demand routing protocol for ad hoc networks. In *MobiCom*, Sept. 2002.

[22] E. Huang, J. Crowcroft, and I. Wassell. Rethinking incentives for mobile ad hoc networks. In *ACM SIGCOMM PINS*, Sept. 2004.

[23] F. Kelly, A. Maulloo, and D. Tan. Rate control in communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research Society*, 49:237–252, 1998.

[24] D. Kotz, C. Newport, and C. Elliott. The mistaken axioms of wireless-network research. Technical Report TR2003-467, Dartmouth, July 2003.

[25] K. Lai, M. Feldman, I. Stoica, and J. Chuang. Incentives for cooperation in peer-to-peer networks. In *Workshop on Economics of Peer-to-Peer Systems*, June 2003.

[26] S. H. Low, F. Paganini, and J. C. Doyle. Internet congestion control. *IEEE Control Systems Magazine*, Feb. 2002.

[27] H. Luo, R. Ramjee, P. Sinha, L. Li, and S. Lu. UCAN: A unified cellular and ad-hoc network architecture. In *MobiCom*, Sept. 2003.

[28] R. Mahajan, S. Floyd, and D. Wetherall. Controlling high-bandwidth flows at the congested router. In *International Conference on Network Protocols (ICNP)*, Nov. 2002.

[29] S. Marti, T. J. Giuli, K. Lai, and M. Baker. Mitigating router misbehavior in mobile ad-hoc networks. In *MobiCom*, Sept. 2000.

[30] J. McClave and F. Dietrich. *Statistics*. Macmillan Publishing Company, 6th edition, 1994.
[31] S. Narayanaswamy, V. Kawadia, R. Sreenivas, and P. R. Kumar. Power control in ad-hoc networks: Theory, architecture, algorithm and implementation of the COMPOW protocol. In *European Wireless Conference*, Feb. 2002.
[32] T.-W. Ngan, D. Wallach, and P. Druschel. Enforcing fair sharing of peer-to-peer resources. In *IPTPS*, Feb. 2003.
[33] M. Pritchard. How to hurt the hackers: The scoop on Internet cheating and how you can combat it. http://www.gamasutra.com/features/20000724/pritchard_pfv.htm, July 2000.
[34] B. Raghavan and A. C. Snoeren. Priority forwarding in ad hoc networks with self-interested parties. In *IPTPS*, June 2003.
[35] MIT Roofnet. http://www.pdos.lcs.mit.edu/roofnet/.
[36] S. Saroiu, K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Multimedia Computing and Networking*, Jan. 2002.
[37] S. Shenker. Making greed work in networks: a game-theoretic analysis of switch service disciplines. *IEEE/ACM Transactions on Networking*, 3(6):819–831, Dec. 2003.
[38] N. Spring, R. Mahajan, and T. Anderson. Quantifying the causes of path inflation. In *ACM SIGCOMM*, Aug. 2003.
[39] V. Srinivasan, P. Nuggehalli, C. F. Chiasserini, and R. R. Rao. Cooperation in wireless ad hoc networks. In *IEEE INFOCOM*, Mar. 2003.
[40] F. Stajano and R. Anderson. The cocaine auction protocol: On the power of anonymous broadcast. In *Int'l Workshop on Information Hiding*, Sept. 1999.
[41] Q. Sun and H. Garcia-Molina. Slic: A selfish link-based incentive mechanism for unstructured peer-to-peer networks. In *24th International Conference on Distributed Computing Systems*, Mar. 2004.
[42] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *SenSys*, Nov. 2003.
[43] J. Zhao and R. Govindan. Understanding packet delivery performance in dense wireless sensor networks. In *SenSys*, Nov. 2003.
[44] S. Zhong, Y. Yang, and J. Chen. Sprite: A simple, cheat-proof, credit-based system for mobile ad hoc networks. In *IEEE INFOCOM*, Mar. 2003.

Notes

[1] Interestingly, "TCP accelerators" have been a concern but have not become pervasive because the bottleneck is usually close to the host, implying that there is little to be gained by deviating from the protocol.

[2] In theory, the cheater can pick an arbitrary signal strength range rather than limiting itself to the top end. But our measurements show that the degree of overlap among neighbors in the middle and bottom part of the range would preclude this behavior. Additionally, better signal strength roughly translates to better connectivity, providing an incentive to pick such neighbors.

A Catch Fail-safes

We briefly consider each step of Catch in light of possible, intentional violations by a free-rider. Our goal is to show that a free-rider cannot defeat the protocol by manipulating messages in unanticipated ways.

Epoch-Start Each node must periodically send EpochStart messages or it is deemed uncooperative by its neighbors and is ignored.

Packet Forwarding and Accounting The testee can drop some or all of the challenges. However, because the challenges are anonymous it: i) cannot selectively inflate the loss rate on some of the links and ii) has to waste its own resources if it chooses to uniformly inflate the loss rate on all links. (Section 3.1)

Anonymous Neighbor Verification Open (ANV1) The testee can drop some fraction of the ANV1 messages. However, this will be detected in a reasonably short time because of anonymity. (Section 3.2)

Tester Information Exchange The testee is unable to interfere with the exchange because it relies on all the testers to release their tokens.

Epoch Evaluation and ANV Close (ANV2) It is in the testee's interest to forward these messages since they are required for it to pass the epoch evaluation.

Isolation Decision Testers drop the free-rider's data packets to isolate it. To prevent this punishment from being circumvented, we require that some unforgeable notion of identity transmitted with data packets.

Deliberate False Accusations A different style of attack is for a *tester* to falsely accuse a cooperative testee and cause it to be isolated. The tester is then no longer required to relay packets for this testee. To discourage this, a cooperative testee retaliates by isolating its accuser, or all of its neighbors, if the identity of the accuser is unknown, i.e., mutually-assured-destruction.

Dropping specific data packets A free-rider can use application-level knowledge to throttle data flow if encryption is not used. For instance, it could selectively drop TCP SYN packets at a higher rate to curb data packet generation. We can detect such behavior by looking for statistical differences in the forwarding rate of such special packets.

Blocking control packets Another possibility is for a node to target specific protocol packets sent by other nodes by interfering with their transmission. This is not plausible because we send protocol packets at randomized times.

Reducing transmission power A free-rider can reduce its relaying responsibilities by reducing its transmission power. This requires the node to be topologically well-placed such that there exists a power level at which it has good connectivity to one other node and almost no connectivity to others. Catch does not counter this strategy, as we view power management to be a legitimate strategy for minimizing co-channel interference.

ACMS: The Akamai Configuration Management System

Alex Sherman[†‡]*, Philip A. Lisiecki*[†]*, Andy Berkheimer*[†]*, and Joel Wein*[†*].
[†]Akamai Technologies, Inc. [‡]Columbia University [*]Polytechnic University.
[†]{*andyb,lisiecki,asherman,jwein*}*@akamai.com*
[‡]*asherman@cs.columbia.edu* [*]*wein@mem.poly.edu.*

Abstract

An important trend in information technology is the use of increasingly large distributed systems to deploy increasingly complex and mission-critical applications. In order for these systems to achieve the ultimate goal of having similar ease-of-use properties as centralized systems they must allow fast, reliable, and lightweight management and synchronization of their configuration state. This goal poses numerous technical challenges in a truly Internet-scale system, including varying degrees of network connectivity, inevitable machine failures, and the need to distribute information globally in a fast and reliable fashion.

In this paper we discuss the design and implementation of a configuration management system for the Akamai Network. It allows reliable yet highly asynchronous delivery of configuration information, is significantly fault-tolerant, and can scale if necessary to hundreds of thousands of servers.

The system is fully functional today providing configuration management to over 15,000 servers deployed in 1200+ different networks in 60+ countries.

1 Introduction

Akamai Technologies operates a system of 15,000+ widely dispersed servers on which its customers deploy their web content and applications in order to increase the performance and reliability of their web sites. When a customer extends their web presence from their own server or server farm to a third party Content Delivery Network (CDN), a major concern is the ability to maintain close control over the manner in which their web content is served. Most customers require a level of control over their distributed presence that rivals that achievable in a centralized environment.

Akamai's customers can configure many options that determine how their content is served by the CDN. These options may include: html cache timeouts, whether to allow cookies, whether to store session data for their web applications among many other settings. Configuration files that capture these settings must be propagated quickly to all of the Akamai servers upon update.

In addition to the configuring customer profiles, Akamai also runs many internal services and processes which require frequent updates or "reconfigurations." One example is the mapping services which assign users to Akamai servers based on network conditions. Subsystems that measure frequently-changing network connectivity and latency must distribute their measurements to the mapping services.

In this paper we describe the Akamai Configuration Management System (ACMS), which was built to support customers' and internal services' configuration propagation requirements. ACMS accepts distributed submissions of configuration information (captured in configuration files) and disseminates this information to the Akamai CDN. ACMS is highly available through significant fault-tolerance, allows reliable yet highly asynchronous and consistent delivery of configuration information, provides persistent storage of configuration updates, and can scale if necessary to hundreds of thousands of servers.

The system is fully functional today providing configuration management to over 15,000 servers deployed in 1200+ different ISP networks in 60+ countries. Further, as a lightweight mechanism for making configuration changes, it has evolved into a critical element of how we administer our network in a flexible fashion.

Elements of ACMS bear resemblance to or draw from numerous previous efforts in distributed systems – from reliable messaging/multicast in wide-area systems, to fault-tolerant data replication techniques, to Microsoft's Windows Update functionality; we present a detailed comparison in Section 8. We believe, however, that our system is designed to work in a relatively unique environment, due to a combination of the following factors.

- The set of end clients – our 15,000+ servers – are very widely dispersed.

- At any point in time a nontrivial fraction of these servers may be down or may have nontrivial connectivity problems to the rest of the system. An individual server may be out of commission for several months before being returned to active duty, and will need to get caught up in a sane fashion.
- Configuration changes are generated from widely dispersed places – for certain applications, any server in the system can generate configuration information that needs to be dispersed via ACMS.
- We have relatively strong consistency requirements. When a server that has been out-of-touch regains contact it needs to become up to date quickly or risk serving customer content in an outdated mode.

Our solution is based on a small set of front-end distributed Storage Points and a back-end process that manages downloads from the front-end. We have designed and implemented a set of protocols that deal with our particular availability and consistency requirements.

The major contributions of this paper are as follows:

- We describe the design of a live working system that meets the requirements of configuration management in a very large distributed network.
- We present performance data and detail some lessons learned from a building and deploying such a system.
- We discuss in detail the distributed synchronization protocols we introduced to manage the front ends Storage Points. While these protocols bear similarity to several previous efforts, they are targeted at a different combination of reliability and availability requirements and thus may be of interest in other settings.

1.1 Assumptions and Requirements

We assume that the configuration files will vary in size from a few hundred bytes up to 100MB. Although very large configuration files are possible and do occur, they in general should be more rare. We assume that most updates must be distributed to every Akamai node, although some configuration files may have a relatively small number of subscribers. Since distinct applications submit configuration files dynamically, there is no particular arrival pattern of submissions, and at times we could expect several submissions per second. We also assume that the Akamai CDN will continue to grow. Such growth should not impede the CDN's responsiveness to configuration changes. We assume that submissions could originate from a number of distinct applications running at distinct locations on the Akamai CDN. We assume that each submission of a configuration file *foo* completely overwrites the earlier submitted version of *foo*. Thus, we do not need to store older versions of *foo*, but the system must correctly synchronize to the latest version. Finally, we assume that for each configuration file there is either a single writer or multiple idempotent (non-competing) writers.

Based on the motivation and assumptions described above we formulate the following requirements for ACMS:

High Fault-Tolerance and Availability. In order to support all applications that dynamically submit configuration updates, the system must operate 24x7 and experience virtually no downtime. The system must be able to tolerate a number of machine failures and network partitions, and still accept and deliver configuration updates. Thus, the system must have multiple "entry points" for accepting and storing configuration updates such that failure of any one of them will not halt the system. Furthermore, these "entry points" must be located in distinct ISP networks so as to guarantee availability even if one of these networks becomes partitioned from the rest of the Internet.

Efficiency and Scalability. The system must deliver updates efficiently to a network of the size of the Akamai CDN, and all parts of the system must scale effectively to any anticipated growth. Since updates, such as a customer's profile, directly effect how each Akamai node serves that customer's content, it is imperative that the servers synchronize relatively quickly with respect to the new updates. The system must guarantee that propagation of updates to all "alive" nodes takes place within a few minutes from submission. (Provided of course, that there is network connectivity to such "alive" or functioning nodes from some of our "entry points.").

Persistent Fault-Tolerant Storage. In a large network some machines will always be experiencing downtime due to power and network outages or process failures. Therefore, it is unlikely that a configuration update can be delivered synchronously to the entire CDN in the time of submission. Instead the system must be able to store the updates permanently and deliver them asynchronously to machines as they become available.

Correctness. Since configuration file updates can be submitted to any of the "entry points," it is possible that two updates for the same file *foo* arrive at different "entry points" simultaneously. We require that ACMS provide a unique ordering of all versions and that the system synchronize to the latest version for each configuration file. Since slight clock skews are possible among our machines, we relax this requirement and show that we allow a very limited, but bounded reordering. (See section 3.4.2).

Acceptance Guarantee. ACMS "accepts" a submis-

sion request only when the system has "agreed" on this version of the update. The agreement in ACMS is based on a "quorum" of "entry points." (The quorum used in ACMS is at the core of our architecture and is discussed in great detail throughout the paper). The agreement is necessary, because if the "entry point" that receives an update submission becomes cut off from the Internet it will not be able to propagate the update to the rest of the system. In essence, the *Acceptance Guarantee* stipulates that if a submission is accepted, a quorum has agreed to propagate the submission to the Akamai CDN.

Security. Configuration updates must be authenticated and encrypted so that ACMS cannot be spoofed nor updates read by any third parties. The techniques that we use to accomplish this are standard, and we do not discuss them further in this document.

1.2 Our Approach

We observe that the ACMS requirements fall into two sets. The first set of requirements deals with update management: highly available, fault-tolerant storage and correct ordering of accepted updates. The second set of requirements deals with delivery: efficient and secure propagation of updates. Instinctively we split the architecture of the system into two subsystems – the "front-end" and the "back-end" – that correspond to the two sets of requirements. The front-end consists of a small set (typically 5 machines) of Storage Points (or SPs). The SPs are deployed in distinct Tier-1 networks inside well-connected data centers. The SPs are responsible for accepting and storing configuration updates. The back-end is the entire Akamai CDN that subscribes to the updates and aids in the update delivery.

High availability and fault-tolerance come from the fact that the SPs constitute a fully decentralized subsystem. ACMS does not depend on any particular SP to coordinate the updates, such as a database master in a persistent MOM (message-oriented middleware) storage. ACMS can tolerate a number of failures or partitions among the Storage Points. Instead of relying on a coordinator, we use a set of distributed algorithms that help the SPs synchronize configuration submissions. These algorithms that will be discussed later are quorum-based and require only a majority of the SPs to stay alive and connected to one another in order for the system to continue operation. Any majority of the SPs can reconstruct the full state of the configuration submissions and continue to accept and deliver submissions.

To propagate updates, we considered a push-based vs. a pull-based approach. In a push-based approach the SPs would need to monitor and maintain state of all Akamai hosts that require updates. In a pull-based approach all Akamai machines check for new updates and request them. We observed that the Akamai CDN itself is fully optimized for HTTP download, making the pull-based approach over HTTP download a natural choice. Since many configuration updates must be delivered to virtually every Akamai server, this allows us to use Akamai caches effectively for common downloads and thus reduce network bandwidth requirements. This natural choice helps ACMS scale with the growing size of the Akamai network.

As an optimization we add an additional set of machines (the Download Points) to the front-end. Download Points offer additional sites for HTTP download and thus alleviate the bandwidth demand placed on the Storage Points.

To further improve the efficiency of the HTTP download we create an index hierarchy that concisely describes all configuration files available on the SPs. A downloading agent can start with downloading the root of the hierarchical index tree and work its way down to detect changes in any particular configuration files it is interested in.

The rest of this paper is organized as follows. We give an architecture overview in section 2. We discuss our distributed techniques of quorum-based replication and recovery in sections 3 and 4. Section 5 describes the delivery mechanism. We share our operational experience and evaluation in sections 6 and 7. Section 8 discusses related work. We conclude in section 9.

2 Architecture Overview

The architecture of ACMS is depicted in Figure 1.

First an application submitting an update (also known as a *publisher*) contacts an ACMS Storage Point. The publisher transmits a new version of a given configuration file. The SP that receives an update submission is also known as the *Accepting SP* for that submission. Before replying to the client the Accepting SP makes sure to replicate the message on at least a quorum (a majority) of Servers (i.e., Storage Points). Servers store the message persistently on disk as a file. In addition to copying the data, ACMS runs an algorithm called Vector Exchange that allows a quorum of SPs to *agree* on a submission. Only after the agreement is reached does the Accepting SP acknowledge the publisher's request, by replying with "Accept."

Once the agreement among the SPs is reached, the data can also be offered for download. The Storage Points *upload* the data to their local HTTP servers (i.e., HTTP servers runs on the same machines as the SPs).

Since only a quorum of SPs is required to reach an agreement on a submission, some SPs may miss an occasional update due to downtime. To account for replication messages missed due to downtime, the SPs run

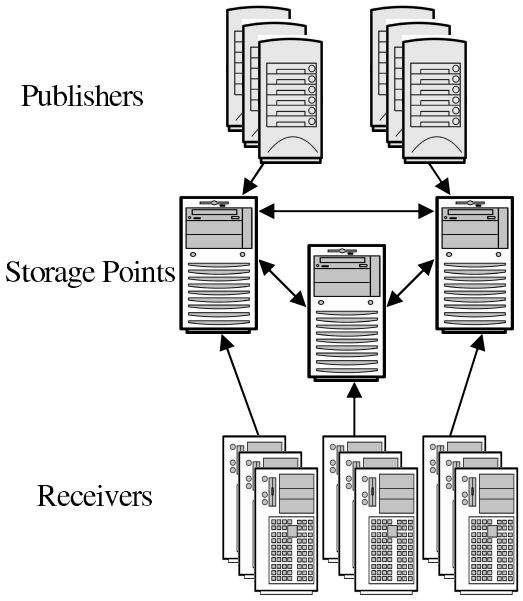


Figure 1: ACMS: Publishers, Storage Points, and Receivers (Subscribers)

a recovery scheme called *Index Merging*. Index Merging helps the Storage Points recover any missed updates from their peers.

To subscribe for configuration updates, each server (also known as a *node*) on the Akamai CDN runs a process called *Receiver* that coordinates subscriptions for that node. Services on each node subscribe with their local Receiver process to receive configuration updates. Receivers periodically make HTTP IMS (If-Modified-Since) requests for these files from the SPs. Receivers send these requests via the Akamai CDN, and most of the requests are served from nearby Akamai caches reducing network traffic requirements.

We add an additional set of a few well-positioned machines to the front-end, called the Download Points (DPs). DPs never participate in initial replication of updates and rely entirely on Index Merging to obtain the latest configuration files. DPs alleviate some of the download bandwidth requirements from the SPs. In this way data replication between the SPs does not need to compete as much for bandwidth with the download requests from subscribers.

3 Quorum-based Replication

The fault-tolerance of ACMS is based on the use of a simple quorum. In order for an Accepting SP to accept an update submission we require that the update be both replicated to and agreed upon by a quorum of the ACMS SPs. We define *quorum* as a majority. As long as a majority of the SPs remain functional and not partitioned from one another, this majority subset will intersect with the initial quorum that accepted a submission. Therefore, this latter subset will collectively contain the knowledge of all previously accepted updates.

This approach is deeply rooted in our assumption that ACMS can maintain a majority of operational and *connected* SPs. If there is no quorum of SPs that are functional and can communicate with one another ACMS will halt and refuse to accept new updates until a *connected quorum* of SPs is re-established.

Each SP maintains connectivity by exchanging liveness messages with its peers. Liveness messages also indicate whether the SPs are fully functional or *healthy*. Each SP reports whether it has pairwise connectivity to a quorum (including itself) of healthy SPs. The reports arrive at the Akamai NOCC (Network Operations Command Center) [2]. If a majority of ACMS SPs fails to report pairwise connectivity to a quorum, a red alert is generated in the NOCC and operation engineers perform immediate connectivity diagnosis and attempt to fix the network or server problem(s).

By placing SPs inside distinct ISP networks we reduce the probability of an outage that would disrupt a quorum of these machines. (See some statistics in section 6.) Since we require only a majority of SPs to be connected, it means we can tolerate a number of failures due to partitioning, hardware, or software malfunctions. For example, with an initial set containing five SPs, we can tolerate two SP failures or partitions and still maintain a viable majority of three SPs. When any single SP malfunctions, a lesser priority alert also triggers corrective action from the NOCC engineers. ACMS operational experience with maintaining a connected quorum and various failure cases are discussed in detail in section 6.

The rest of the section describes the quorum-based Acceptance Algorithm in detail. We also explain how ACMS replication and agreement methods satisfy *Correctness* and *Acceptance* requirements outlined in section 1.1 and discuss maintenance of the ACMS SPs.

3.1 Acceptance Algorithm

The ACMS Acceptance Algorithm consists of two phases: replication and agreement. In the replication phase, the Accepting SP copies the update to at least a quorum of the SPs.

The Accepting SP first creates a temporary file with a unique filename (UID). For a configuration file *foo* the UID may look like this: "foo.A.1234", where *A* is the name of the Accepting SP and "1234" is the timestamp of the request in UTC (shortened to 4 digits for this example). This UID is unique, because each SP allows only

one request per file per second.

The Accepting SP then sends this file that contains the update along with its MD5 hash to a number of SPs over a secure TCP connection. Each SP that receives the file stores it persistently on disk (under the UID name), verifies the hash, and acknowledges that it has stored the file.

If the Accepting SP fails to replicate the data to a quorum after a timeout, it replies with an error to the publishing application. The timeout is based on the size of the update, and a very low estimate of available bandwidth between this SP and its peers. (If the Accepting SP does not have connectivity to a quorum it replies much sooner and does not wait for a timeout to expire).

Otherwise, once at least a quorum of SPs (including the Accepting SP) has stored the temporary file, the Accepting SP initiates the second phase to obtain an agreement from the Storage Points on the submitted update.

3.2 Vector Exchange

Vector Exchange (also called "VE") is a light-weight protocol that forms the second phase of the acceptance algorithm – the *agreement* phase. As the name suggests, VE involves Storage Points exchanging a state vector. The VE vector is just a bit vector with a bit corresponding to each Storage Point. A 1-bit indicates that the corresponding Storage Point *knows* of a given update. When a majority of bits are set to 1, we say that an *agreement* occurs and it is safe for any SP (that sees the majority of the bits set) to upload this latest update.

In the beginning of the agreement phase, the Accepting SP initializes a bit vector by setting its own bit to 1 and the rest to 0, and broadcasts the vector along with the UID of the update to the other SPs. Any SP that sees the vector sets its corresponding bit to 1, stores the vector persistently on disk and re-broadcasts the modified vector to the rest of the SPs. Persistent storage guarantees that the SP will not lose its vector state on process restart or machine reboot. It is safe for each SP to set the bit even if it did not receive the temporary file during the replication phase. Since at least a quorum of the SPs have stored this temporary file, it can always locate this file at a later stage.

Each SP learns of the agreement independently when it sees a quorum of bits set. Two actions can take place when a SP learns of the agreement for the first time. When the Accepting SP that initiated the VE instance learns of the agreement it *accepts* the submission of the publishing application. When any SP (including the Accepting SP) learns of the agreement it *uploads* the file. Uploading means that the SP copies the temporary file to a permanent location on its local HTTP server where it is now available for download by the Receivers. If it does not have the temporary file then it downloads it from one of the other SPs via the recovery routine (section 4).

Note, that it is possible for the Accepting SP to become "cut-off" from the quorum after it initiates the VE phase. In this case it does not know whether its broadcasts were received and whether the agreement took place. It is then forced to reply only with "Possible Accept" rather than "Accept" to the publishing application. We recommend that the publisher that gets cut off from the Accepting SP or receives a "Possible Accept" should try to re-submit its update to another SP. (From a publisher's perspective the reply of "Possible Accept" is equivalent to "Reject." The distinction was made initially purely for the purpose of monitoring this condition.)

As in many *agreement* schemes, the purpose of the VE protocol is to deal with some Byzantine network or machine failures [18]. In particular, VE prevents an individual SP (or a minority subset of SPs) from uploading new data and then becoming "disconnected" from the rest of the SPs. A quorum of SPs could then continue to operate successfully without the knowledge that the minority is advertising a new update. This new update would become available only to a small subset of the Akamai nodes that can reach the minority subset, possibly causing a discord in the Akamai network viz. the latest updates.

VE is based on earlier ideas of vector clocks introduced by by Fidge [10] and Mattern [24]. Section 8 compares Acceptance Algorithm with Two-Phase Commit and other agreement schemes used in common distributed systems.

3.3 An Example

We give an example to demonstrate both phases of the Acceptance Algorithm. Imagine that our system contains five Storage Points named *A*, *B*, *C*, *D*, and *E* with SP *D* down temporarily for a software upgrade. With five SPs the quorum required for the Acceptance algorithm is three SPs.

SP *A* receives a submission update from publisher *P* for configuration file "foo". To use the example from section 3.1 SP *A* stores the file under a temporary UID: *foo.A.1234*.

SP *A* initiates the replication phase by sending the file in parallel to as many SPs as it can reach. SPs *B*, *C*, and *E* store the temporary update under the UID name. (SP D is down and does not respond). SPs *B* and *C* happen to be the first SPs to acknowledge the reception of the file and the MD5 hash check. Now *A* knows that the majority (*A*, *B*, and *C*) have stored the file and *A* is ready to initiate the agreement phase.

SP *A* broadcasts the following VE message to the other SPs:

```
foo.A.1234 A:1 B:0 C:0 D:0 E:0
```

This message contains the UID of the pending update and the vector that has only *A*'s bit set. (*A* stores this vector state persistently on disk prior to sending it out).

When SP *B* receives this message it adds its bit to the vector, stores the vector, and broadcasts it:

```
foo.A.1234 A:1 B:1 C:0 D:0 E:0
```

After a couple of rounds all four live SPs store the following message with all bits set except for *D*'s:

```
foo.A.1234 A:1 B:1 C:1 D:0 E:1
```

At this point, as each SP sees that the majority of bits is set, *A*, *B*, *C*, and *E* upload the temporary file in place of the permanent configuration file foo, and store in their local database the UID of the latest agreed upon version of file foo: *foo.A.1234*. All older records of foo can be discarded.

3.4 Guarantees

We now show that our Acceptance Algorithm satisfies the acceptance and correctness requirements, provided that our quorum assumption holds.

3.4.1 Acceptance Guarantee

Having introduced the quorum-based scheme we now restate the acceptance guarantee more precisely than in section 1.1. The acceptance guarantee states that if the Accepting SP has accepted a submission, it will be uploaded by a quorum of SPs.

Proof: The Accepting SP accepts only when the update has been replicated to a quorum AND when the Accepting SP can see a majority of bits set in the VE vector. Now if the Accepting SP can see a majority of bits set in the VE vector it means that at least a majority of the SPs have stored a partially filled VE vector during the agreement phase. Therefore, any future quorum will include at least one SP that stores the VE vector for this update. Once such a SP is part of a quorum, after a few re-broadcast rounds, all of the SPs in this future quorum will have their bits set. Therefore, all the SPs in the latter quorum will decide to upload.

So based on our assumption that a quorum of connected SPs can be reasonably maintained, acceptance by ACMS implies a future decision by at least a quorum to upload the update.

The converse of the acceptance guarantee does not necessarily hold. If the quorum decides to upload, it does not mean that the Accepting SP will accept. As stated earlier the Accepting SP may be "cut off" from the quorum after VE phase is initiated, but before it completes. In that case the Accepting SP replies with "Possible Accept," because it's likely but not definite. The publishing application treats this reply as "Reject" and tries to resubmit to another SP.

The probability of a "Possible Accept" is very small, and we have never seen it occur in the real system. The reason for that is that in order for the VE phase to be initiated the replication phase must succeed. If the replication is successful it most likely means that the lighter VE phase that also requires connectivity to a quorum (but less bandwidth) will also succeed. If the replication phase fails, ACMS replies with a definite "Reject."

3.4.2 Correctness

The Correctness requirements state that ACMS provides a unique ordering of all update versions for a given configuration file AND that the system synchronizes to the latest submitted update. We later relaxed that guarantee to state that ACMS allows *limited* re-ordering in deciding which update is the latest, due to clock skews. More precisely, accepted updates for the same file submitted at least $2T + 1$ seconds apart will be ordered correctly. T is the maximum allowed clock skew between any two communicating SPs.

The unique ordering of submitted updates is guaranteed by the UID assigned to a submission as soon as it is received by ACMS (regardless of whether it will be accepted). The UID contains both a UTC timestamp from the SP's clock and the SP's name. The submissions for the same configuration file are first ordered by time and then by the Accepting SP name. So "foo.B.1234" is considered to be more recent than "foo.A.1234", and it is kept as the later version. A Storage Point accepts only one update per second for a given configuration file.

Since we do not use logical synchronized clocks, slight clock skews and reordering of updates are possible. We now explain how we bound such reordering, and why any small reordering is acceptable in ACMS.

We bound the possible skew between any two communicating SPs by T seconds (where T is usually set to 20 seconds). Our communication protocols enforce this bound by rejecting liveness messages from SPs that are at least T seconds apart. (I.e., such pairs of servers appear virtually dead to each other). As a result it follows that no two SPs that accept updates for the same file can have a clock skew more than $2T$ seconds.

Proof: Imagine SPs *A* and *B* that are both able to accept updates. This means both *A* and *B* are able to replicate these update to a majority of SPs. These majorities must overlap by at least one SP. Moreover, neither *A* nor *B* can have more than a T second clock skew from that SP. So *A* and *B* cannot be more than $2T$ seconds apart.

Developers of the Akamai subsystems that submit configuration files to Akamai nodes via ACMS are advised to avoid mis-ordering by submitting updates to the

same configuration file at intervals of at least $2T + 1$. In addition, we use NTP [3] to synchronize our server clocks, and in practice we find very rare instances when our servers are more than one second apart.

Finally with ACMS, it is actually acceptable to reorder updates within a small bound such as $2T$. We are not dealing with competing editors of a distributed filesystem. Subsystems that are involved in configuring a large CDN such as Akamai must and do cooperate with each other. In fact, we considered two cases of such subsystems that update the same configuration file. Either there is only one process that submits updates for file "foo", or there are redundant processes that submit the same or idempotent updates for file "foo". In the case of a single publishing process, it can easily abide by the $2T$ rule and therefore avoid reordering. In the case of redundant writers – that exist for fault-tolerance – we do not care whose update within the $2T$ period is submitted first as these updates are idempotent. Any more complex distributed systems that publish to ACMS use leader election to select a publishing process, effectively reducing these systems to one-publisher systems.

3.5 Termination and Message Complexity

In almost all cases VE terminates after the last SP in a quorum broadcasts its addition to the VE vector. However, in an unlikely event where a SP becomes partitioned off during a VE phase it attempts to broadcast its vector state once every few seconds. This way, once it reconnects to a quorum it can notify the other SPs of its partial state.

VE is not expensive and the number of messages exchanged is quite small. We make a small change to the protocol as it was originally described by adding a small random delay (under 1 second) before a re-broadcast of the changed vector by a SP. This way, instead of all SPs re-broadcasting in parallel, only one SP broadcasts at a time. With the random delay, on average each SP will only broadcast once after setting its bit. This results in $O(n^2)$ unicast messages.

We use the gossip model, because the numbers of participants and the size of the messages are both small. The protocol can easily be amended to have only the Accepting SP do a re-broadcast after it collects the replies. Only when an SP does not hear the re-broadcast does it switch to a gossip mode. When the Accepting SP stays connected until termination the number of messages exchanged is just $O(n)$.

3.6 Maintenance

Software or OS upgrades performed on individual Storage Points must be coordinated to prevent an outage of a quorum. Such upgrades are scheduled independently on individual Storage Points so that the remaining system still contains a connected quorum.

Adding and removing machines with quorum-based systems is a theoretically tricky problem. Rambo [19] is an example of a quorum-based system that solves dynamic set configuration changes by having an old quorum agree on a new configuration.

Since adding or removing SPs is extremely rare we chose not to complicate the system to allow dynamic configuration changes. Instead, we halt the system temporarily by disallowing accepts of new updates, change the set configuration on all machines, wait for a new quorum to sync up on all state (via the Recovery algorithm), and allow all SPs to resume operation. Replacing a dead SP is a simpler procedure where we bring up a new SP with the same SP ID as the old one and clean state.

3.7 Flexibility of the VE Quorum

ACMS' quorum is configured as *majority*. Just like in the Paxos [16] algorithm this choice guarantees that any future quorum will necessarily intersect with an earlier one and all previously accepted submissions can be recovered. However, this definition is quite flexible in VE and allows for consistency vs. availability trade-offs. For example, one could define a quorum to be just a couple of SPs which would offer loose consistency, but much higher availability. Since there is a new VE instance for each submission, one could potentially configure a different quorum for each file. If desired, this property can be used to add or remove SPs by reconfiguring each SP independently, resulting in a very slight and temporary shift toward consistency over availability.

4 Recovery via Index Merging

Recovery is an important mechanism that allows all Storage Points that experience down time or a network outage to "sync up" all latest configuration updates.

Our Acceptance Algorithm guarantees that at least a quorum of SPs stores each update. Some Akamai nodes may only be able to reach a subset of the SPs that were not part of the quorum that stored the update. Even if that subset intersects with the quorum, that Akamai node may need to retry multiple downloads before reaching a SP that stores the update. To increase the number of Akamai nodes that can get their updates and improve the efficiency of download, preferably all SPs should store all state.

In order to "sync up" any missed updates Storage Points continuously run a background recovery protocol with one another. The downloadable configuration files are represented on the SPs in the form of an index tree.

The recovery protocol is called Index Merging. The SPs "merge" their index trees to pick up any missed updates from one another.

The Download Points also need to "sync up" state. These machines do not participate in the Acceptance Algorithm and instead rely entirely on the recovery protocol on Storage Points to pick up all state.

4.1 The Index Tree

For a concise representation of the configuration files, we organize the files into a tree. The configuration files are split into groups. A Group Index file lists the UIDs of the latest agreed upon updates for each file in the group. The Root Index file lists all Group Index files together with the latest modification timestamps of those indexes. The top two layers (i.e. the Root and the Group indexes) completely describe the latest UIDs of all configuration files and together are known as the *snapshot* of the SP.

Each SP can modify its *snapshot* when it learns of a quorum agreement through the Acceptance Algorithm or by seeing a more recent UID in a snapshot of another SP.

Since a quorum of SPs should together have a complete state, for full recovery each SP needs only to merge in a snapshot from $Q - 1$ other SPs (where $Q = majority$). (Download Points need to merge in state from Q SPs).

The configuration files are assigned to groups statically when the new configuration file is provisioned on ACMS. A group usually contains a logical set of files subscribed to by a set of related receiving applications.

4.2 The Index Merging Algorithm

At each round of the Index Merging Algorithm a SP A picks a random set of $Q - 1$ other SPs and downloads and parses the index files from those SPs. If it detects a more recent UID of a configuration file, SP A updates its own snapshot, and attempts to download the missing file from one of its peers. Note that it is safe for A to update its snapshot before obtaining the file. Since the UID is present in another SP's snapshot it means that the file has already been agreed upon and stored by a quorum.

To avoid frequent parsing of one another's index files, the SPs remember the timestamps of one another's index trees and make HTTP IMS (if-modified-since) requests. If an index file has not been changed, HTTP 304 (not-modified) is returned on the download attempt.

Index Merging rounds run continuously.

4.3 Snapshots for Receivers

As a side-effect the snapshots also provide an efficient way for Receivers to learn of latest configuration file versions. Typically receivers are only interested in a subset of the index tree that describes their subscriptions. Receivers also download index files from the SPs via HTTP IMS requests.

Using HTTP IMS is efficient but is also problematic because each SP generates its own snapshot and assigns its own timestamps to the index files that it uploads. Thus it is possible for a SP A to generate an index file with more recent timestamp than SP B, but less recent information. If a Receiver is unlucky and downloads the index file from A first, it will not download an index with a lower timestamp from B, until the timestamp increases. It may take a while for it to get all the necessary changes.

There are two solutions to this problem. In one solution we could require a Receiver to download an index tree independently from each SP, or at least a quorum of the SPs. Having each Receiver download multiple index trees is an unnecessary waste of bandwidth. Furthermore, requiring each Receiver to be able to reach a a quorum of SPs reduces system availability. Ideally, we only require that a Receiver be able to reach one SP that itself is part of a quorum.

We implemented an alternative solution, where the SPs merge their index timestamps, not just the data listed in the those indexes.

4.4 Index Time-stamping Rules

With just a couple of simple rules that constrain how Storage Points assign timestamps to their index files, we can present a coherent snapshot view to the Receivers:

1. If a Storage Point *A* has an index file bar.index with a timestamp T, and then *A* learns of new information inside bar.index (either through Vector Exchange agreement or Index Merging from a peer), then on the next iteration *A* must upload a new bar.index with a timestamp at least $T + 1$.

2. If Storage Points *A* and *B* have an index file bar.index that contains identical information and have timestamps T_a and T_b respectively with $T_a > T_b$, then on the next iteration *B* must upload bar.index with a timestamp at least as great as T_a.

Simply put, rule 1 says that when a Storage Point includes new information it must increase the timestamp. This is really a redundant rule — a new timestamp would be assigned anyway when a Storage Point writes a new file. Rule 2 says that a Storage Point should always set its index's timestamp to the highest timestamp for that index among its peers (even if it includes no new information).

Once a Storage Point modifies a group index it must modify the Root Index as well following the same rules. (The same would apply to a hierarchy with more layers).

We now show the correctness of this timestamping algorithm.

4.5 Timestamping Correctness

Guarantee: If a Receiver downloads bar.index (index file for group bar) with a timestamp T_1 from any Storage Point, then when new information in group bar becomes available all Storage Points will publish bar.index with a timestamp at least as big as $T_1 + 1$, so that the Receiver will quickly pick up the change.

Proof: Assume in steady state a set of k Storage Points $1...k$ each has a bar.index with timestamps $T_1, T_2, ..., T_k$ sorted in non-decreasing order. (i.e., T_k is the highest timestamp). When new information becomes available, then following rule 1 above, Storage Point k will incorporate new information and increase its timestamp to at least $T_k + 1$. On the next iteration, following rule 2, SPs $1...k-1$ will make their timestamps at least $T_k + 1$ as well. Before the change, the highest timestamp for bar.index known to a Receiver was T_k. A couple of iterations after the new information becomes incorporated, the lowest timestamp available on any Storage Point is $T_k + 1$. Thus, a Receiver will be able to detect an increase in the timestamp and pick up a new index quickly.

5 Data Delivery

In addition to providing high fault-tolerance and availability the system must scale to support download by thousands of Akamai servers. We naturally use the Akamai CDN (Content Distribution Network) which is optimized for file download. In this section we describe the Receiver process, its use of the hierarchical index data, and the use of the Akamai CDN itself.

5.1 Receiver Process

Receivers run on each of over 15,000 Akamai nodes and check for message updates on behalf of the local subscribers.

A Subscription for a configuration file specifies the location of that file in the index tree: the root index, the group index that includes that file, and the file name itself. Receivers combine all local subscriptions into a subscriptions tree. (This is a subtree of the whole tree stored by the SPs.)

A Receiver checks for updates to the subscription tree by making HTTP IMS requests recursively beginning at the Root Index. If the Root Index has changed, Receiver parses the file, and checks whether any intermediate indexes that are also in the Receiver's subscription tree have been updated (i.e., if they are listed with a higher timestamp than previously downloaded by that Receiver). If so, it stores the timestamp listed for that index as the "target timestamp," and keeps making IMS requests until it downloads the index that is at least as recent as the target timestamp. Finally it parses that index and checks whether any files in its subscription tree (that belong to this index) have been updated. If so the Receiver then tries to download a changed file until it gets one at least as recent as the target timestamp.

There are a few reasons why a Receiver may need to attempt multiple IMS requests before it gets a file with a target timestamp. First some Storage Points may be a bit behind with Index Merging and not contain the latest files. Second, an old file may be cached by the Akamai network for a short while. The Receiver retries its downloads frequently until it gets the required file. Once the Receiver downloads the latest update for a subscription, it places the data in a file on local disk and points a local subscriber to it.

The Receiver must know how to find the SPs. The Domain Name Service provides a natural mechanism to distribute the list of SPs' and DPs' addresses.

5.2 Optimized Download

The Akamai network's support for HTTP download is a natural fit to be leveraged by ACMS for message propagation. Since the indexes and the configuration files are requested by many machines on the network, these files benefit greatly from the caching capabilities of the Akamai network.

First, Receivers running on colocated nodes are likely to request the same files, which makes it likely that the request is served from a neighboring cache in the local Akamai cluster. Furthermore, if the request leaves the cluster it will be directed to other nearby Akamai clusters which are also likely to have a response cached. Finally, if the file is not cached in another nearby Akamai cluster, the request goes through to one of the Storage Points. These cascading Akamai caches greatly reduce the network bandwidth required for message distribution and make pull-down propagation the ideal choice.

The trade-off of having great cacheability is the increased propagation delay of the messages. The longer the file is served out of cache, the longer it takes for the Akamai system to refresh cached copies. Since we are more concerned here with efficient rather than very fast delivery, we set a long cache TTL on the ACMS files, for example, 30 seconds.

As mentioned in section 2 we augment the list of SPs with a set of a few Download Points. Download Points provide an elegant way to alleviate bandwidth requirements from the SPs. As a result replication and recovery algorithms on the SPs experience less competition with the download bandwidth.

6 Operational Experience

The design of ACMS has been an iterative process between implementation and field experience where our assumptions of persistent storage, network connectivity, and OS/software fault-tolerance were tested.

6.1 Earlier Front-End Versions

Our prototype version of ACMS consisted of a single primary Accepting Storage Point replicating submissions to a few secondary Storage Points. Whenever the Accepting SP would lose connectivity to some of the Storage Points or experience a software or hardware malfunction the entire system would halt. It quickly became imperative to design a system that did not rely entirely on any single machine. We also considered a solution of using a set of auto-replicating databases. We encountered two problems. First, commercial databases would prove unnecessarily expensive as we would have to acquire licenses to match the number of customers using ACMS. More importantly, we required consistency. At the time we did not find database software that would deal with various Byzantine network failures. Although some academic systems were emerging that in theory did promise the right level of wide-area fault-tolerance we required a professional, field-tested system that we could easily tune to our needs. Based on our study of Paxos [16] and BFS [17] we designed a simpler version of decentralized quorum-based techniques. Similar to Paxos and BFS our algorithm requires a quorum. However, there is no leader to enforce strict ordering in VE as bounded re-ordering is permitted with non-competing configuration applications.

6.2 Persistent Storage Assumption

Storage Points rely on persistent disk storage to store configuration files, snapshots, and temporary VE vectors. Most hard disks are highly reliable, but guarantees are not absolute. Data may get corrupted, especially on systems with high levels of I/O. Moreover, if the operating system crashes before an OS buffer is flushed to disk, the result of the write may be lost.

After experiencing a few file corruptions we adopted the technique of writing out MD5 hash together with the file's contents before declaring a successful write. The hash is checked on opening the file. A Storage Point which detects a corrupted file will refuse to communicate with its peers and require an engineer's attention. Over the period of six months ending in February 2005, the NOCC [2] monitoring system has recorded 3 instances of such file corruption on ACMS.

Since ACMS runs automatic recovery routines replacing damaged or old hardware on ACMS is trivial. The SP process running on a clean disk quickly recovers all of the ACMS state from other SPs via Index Merging.

6.3 Connected Quorum Assumption

The assumption of a connected quorum turned out to be a very good one. Nonetheless, network partitions do occur, and the quorum requirement of our system does play its role. For the first 9 months of 2004 the NOCC monitoring system recorded 36 instances where a Storage Point did not have connectivity to a quorum due to network outages that lasted for more than 10 minutes. However, in all of those instances there was an operating quorum of other SPs that continued to accept submissions.

Brief network outages on the Internet are also common although they would generally not result in a SP losing connectivity to a quorum. For example, a closer analysis of ACMS logs over a 6 day period revealed two short outages within the same hour between a pair of SPs located in different Tier-1 networks. They lasted for 8 and 2 minutes respectively. Such outages emphasize the necessity for an ACMS-like design to provide uninterrupted service.

6.4 Lessons Learned

As we anticipated, redundancy has been important in all aspects of our system. Placing the SPs in distinct networks has protected ACMS from individual network failures. Redundancy of multiple replicas helped ACMS cope with disk corruption and data loss on individual SPs.

Even the protocols used by ACMS are in some sense redundant. The continuous recovery scheme (i.e., Index Merging) helps the Storage Points recover updates that they may miss during the initial replication and agreement phases of the Acceptance Algorithm. In fact, in some initial deployments Index Merging helped ACMS overcome some communication software glitches of the Acceptance Algorithm.

The back-end of ACMS also benefited from redundancy. Receivers begin their download attempt from nearby Akamai nodes, but can fail over to higher layers of the Akamai network if needed. This approach allows Receivers to cope with downed servers on their download path.

Despite the redundant and self-healing design sometimes human intervention is required. We rely heavily on the Akamai error reporting infrastructure and the operations of the NOCC to prevent critical failures of ACMS. Detection of and response to secondary failures such as

individual SP corruption or downtime helps decrease the probability of full quorum failures.

7 Evaluation

To evaluate the effectiveness of the system we gathered data from the live ACMS system accepting and delivering configuration updates on the actual Akamai network.

7.1 Submission and Propagation

First we looked at the workload of the ACMS front-end over a 48 hour period in the middle of a work week. There were 14,276 total file submissions on the system with five operating Storage Points. The table below lists the distribution of the file sizes. Submission of smaller files (under 100KB) were dominant, but files on the order of 50MB also appear about 3% of the time.

size range	avg file sz	distribution	avg.time (s)
0K-1K	290	40%	0.61
1K-10K	3K	26%	0.63
10K-100K	22K	23%	0.72
100K-1M	167K	7%	2.23
1M-10M	1.8M	1%	13.63
10M-100M	51M	3%	199.87

The last column of the table shows the average submission time for various file sizes. We evaluated the "submission" time by measuring the period from the time an Accepting SP is first contacted by a publishing application, until it replies with "Accept." The submission time includes replication and agreement phases of the Acceptance Algorithm. The agreement phase for all files takes 50 milliseconds on average. For files under 100KB, all "submission" times are under one second. However, with larger files, replication begins to dominate. For example, for 50MB files, the time is around 200 seconds. Even though our SPs are located in Tier 1 networks they all share replication bandwidth with the download bandwidth from the Receivers. In addition, replication for multiple submissions and multiple peers is performed in parallel.

We also measured the total update propagation time from when many configuration updates were first made available for download through receipt on the live Akamai network for a random sampling of 250 Akamai nodes. Figure 2 shows the distribution of update propagation times. The average propagation time is approximately 55 seconds. Most of the delay comes from Receiver polling intervals and caching.

Figure 3 examines the effect of file size on propagation time. We have analyzed the mean and 95th-percentile delivery time for each submission in the test period. 99.95% of updates arrived within three minutes. The remaining 0.05% were delayed due to temporary network

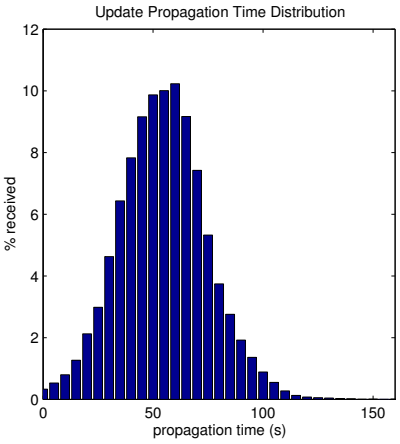


Figure 2: Propagation time distribution for a large number of configuration updates delivered to a sampling of thousands of machines.

connectivity issues; the files were delivered promptly after connectivity was restored. These delivery times meet our objectives of distributing files within several minutes. The figure shows a high propagation time for especially small files. Although one would expect that the propagation time increases monotonically with the file size, CDN caching slows down files submitted more frequently. We believe that many smaller files are updated frequently on ACMS. As a result the caching TTL of the CDN is more heavily reflected in propagation delay.

The use of caching reduces bandwidth on the Storage Points anywhere from 90% to 99%, increasing in general with system activity and with the file size being pushed, allowing large updates to be propagated to tens of thousands of machines without significant impact on Storage Point traffic.

Finally to analyze general connectivity and the tail of the propagation distribution we looked at a propagation of short files (under 20KB) to another random sample of 300 machines over a 4 day period. We found that 99.8% of the time a file was received within 2 minutes from becoming available and 99.96% of the time it was received within 4 minutes.

7.2 Scalability

We analyzed the overhead of the Acceptance Algorithm and its effect on the scalability of the front-end. Over a recent 6 day period we recorded 43,504 successful file submissions with an average file size of 121KB. In a system with 5 SPs, the Accepting SP needs to replicate data to 4 other SPs requiring 484 KBytes per file on average. The size of a VE message is roughly 100 bytes. With $n(n-1)$ VE messages exchanged per submission, VE uses 2 KB per file or 0.4% of the replication bandwidth.

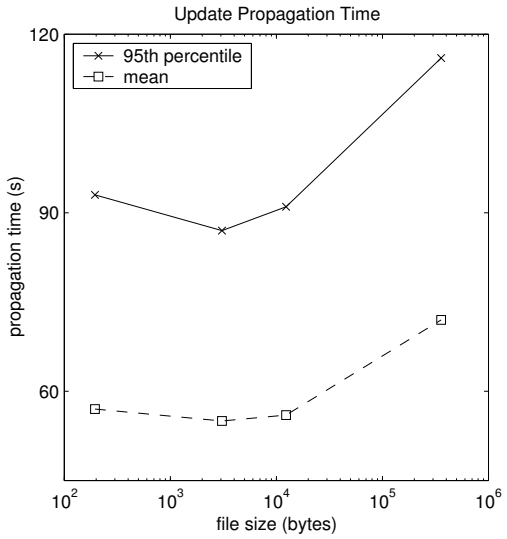


Figure 3: Propagation times for various size files. The dashed line shows the average time for each file to propagate to 95% of its recipients. The solid line shows the average propagation time.

For our purposes we chose 5 SPs, so that during a software upgrade of one machine the system cold tolerate one failure and still maintain a majority quorum of 3. Extending the calculation to 15 SPs, for example, with an average file size of 121 KB the system would require 1.7 MB for replication and 21KB for VE. The VE overhead becomes 1.2%, which is higher, but not significant.

Such a system is conceivable if one chooses not to rely on a CDN for efficient propagation, but instead offer more download sites (SPs). The VE overhead can be further reduced as described in section 3.5. However, the minimum bandwidth required to replicate the data to all 15 machines may grow to be prohibitive. In such a system one could still allow each Server to maintain all indexes, but split the actual storage into subsets based on some hashing function such as Consistent Hashing [4].

For ACMS choosing the Akamai CDN itself for propagation is the natural choice. The cacheability of the system grows as the CDN penetrates more ISP networks, and the system scales naturally with its own growth. Also, as the CDN grows the reachability of receivers inside more remote ISPs improves.

8 Related Work

8.1 Fault Tolerant Replication

Many distributed filesystems such as Coda [20], Pangea [21], and Bayou [22] store files across multiple replicas similar to the Storage Points of ACMS. Similar to ACMS' Index Merging these filesystems run recovery algorithms that synchronize the data among replicas, such as Bayou's anti-entropy algorithm. However, all of these systems attempt to improve the availability of data at the expense of consistency. The aim is to allow file operations to clients on a set of disconnected machines. ACMS, on the other hand must provide a very high level of consistency across the Akamai network and cannot allow a single SP to accept and upload a new update independently.

The two-phase Acceptance Algorithm used by ACMS is similar in nature to the Two Phase-Commit [12]. Two-phase commit also separates a transaction phase from a commit phase, but its failure modes make it more suitable to a local environment.

The Vector Exchange (the agreement phase of our algorithm) was inspired by the concept of vector clocks introduced by Fidge [10] and Mattern [24] which are used to determine causality of events in a distributed system. Bayou also uses vectors to represent latest known commit sequence numbers for each server. In our algorithm, the vectors' contents are simply bits since each message only has two interesting states, known to a server or not. Each subsequent agreement is a separate "instance" of the protocol.

VE uses a quorum-based scheme similar to Paxos [16] and BFS [17]. Paxos defines quorum as strict majority while BFS defines it as "more than $2/3$." VE allows "quorum" to be configurable as long as it is at least a majority. All these algorithms consider Byzantine failures and rely on persistent storage by a quorum to enable a later quorum to recover state. This strong property precludes scenarios allowed by a simpler two phase commit protocol for a minority of partitioned replicas to commit a transaction. Other quorum systems include weighted voting [11] and hierarchical quorum consensus [15].

At the same time VE is simpler than Paxos and BFS and does not implement a full Byzantine Fault-Tolerance. It does not require an auxiliary protocol to determine a leader or a *primary* as in Paxos or BFS respectively. This relaxation stems from the nature of ACMS applications where only a single or redundant writers exist for each file and thus, some bounded reordering is permissible as explained in section 3.4.2. No leader is enforcing ordering.

OceanStore [31] is an example of a storage system that implements Byzantine Fault-Tolerance to have replicas agree on the order of updates that originate from different sources. ACMS, on the other hand must complete "agreement" at the time of an update submission. This is primarily due to the important aspect of the Akamai network where an application that publishes a new configuration file must know that the system has agreed to upload and propagate the new update. (Otherwise it will

keep retrying.)

8.2 Data Propagation

Similar to multicast [9], ACMS is designed to deliver data to many widely dispersed nodes in a way that conserves bandwidth. While ACMS takes advantage of the Akamai Network optimizations for hierarchical file caching, multicast uses proximity of network IP addresses to send fewer IP packets. However, due to the lack of more intelligent routing infrastructure between major networks on the Internet, it is virtually impossible to multicast data across these networks.

To bypass the Internet routing shortcomings many application-level multicast schemes based on overlay networks were proposed: CAN-Multicast [27], Bayeux [34], and Scribe [29] among others [14] [7]. These systems leverage communication topologies of P2P overlays such as CAN [26], Chord [30], Pastry [28], Tapestry [33]. Unlike ACMS, these systems create a propagation tree for each new source of the multicast, incurring an overhead. As shown in [5], using these systems for multicast is not always efficient. In our system on the other hand, once the data is injected into ACMS, it is available for download from any Storage or Download Point, and propagates down the tree from these distinct well-connected sources. The effect of the overlay networks used in reliable multicasting networks [23], [6] is replaced by cooperating caches in our system.

ACMS is similar to Messaging Oriented Middleware (MOM) in that it provides persistent storage and asynchronous delivery of updates to subscribers that may be temporarily unavailable. Common MOMs include Sun's JMS [32], IBM's MQSeries [13], Microsoft's MSMQ [25], and the like. These system usually contain a server that persists the messaging "queue" which helps deal with crash recovery, but does create a single point of failure. The distributed model of ACMS storage, on the other hand, helps it tolerate multiple failures or partitions.

8.3 Software Updates

Finally, we compare a complete ACMS with existing software update systems. LCFG [35] and Novadigm [36] create systems to manage desktops and PDAs across an enterprise. While these systems scale to thousands of servers they usually span a single or a few enterprise networks. ACMS, on the other hand delivers updates across multiple networks for critical customer-facing applications. As a result ACMS focuses on a highly fault-tolerant storage and efficient propagation.

Systems that deliver software, like Windows Updates [37] target a much larger set of machines than found in the Akamai network. However, polling intervals for such updates are not as critical. Some Windows users take days to activate their updates while each Akamai node is responsible for serving requests to tens of thousands of users and thus must synchronize to the latest updates very efficiently. Moreover, systems such as Windows Updates use a rigorous, centralized process to push out new updates. ACMS accepts submissions from dynamic publishers dispersed throughout the Akamai network. Thus, highly fault-tolerant, available, and consistent storage of updates is required.

9 Conclusion

In this paper we have presented the Akamai Configuration Management System that successfully manages configuration updates for the Akamai network of 15,000+ nodes. Through the use of simple quorum-based algorithms (Vector Exchange and Index Merging), ACMS provides highly available, distributed, and fault-tolerant management of configuration updates. Although these algorithms are based on earlier ideas, they were particularly adapted to suit a configuration publishing environment and provide high level of consistency and easy recovery for the ACMS' Storage Points. These schemes offer much flexibility and may be useful in other distributed systems.

Just like ACMS, any other management system could benefit from using a CDN such as Akamai's to propagate updates. First, a CDN managed by a third party offers a convenient overlay that can span thousands of networks effectively. A solution such as multicast requires much management and simply does not scale across different ISPs. Second, a CDN's caching and reach will allow the system to scale to hundreds of thousands of nodes and beyond.

Most importantly we have presented valuable lessons learned from our operational experience. Redundancy of machines, networks, and even algorithms helps a distributed system such as ACMS cope with network and machine failures, and even human errors. Despite 36 network failures that we recorded in the last 9 months, that affected some ACMS Storage Points, the system continued to operate successfully. Finally, active monitoring of any critical distributed system is invaluable. We relied heavily on the NOCC infrastructure to maintain a high level of fault-tolerance.

Acknowledgements

We would like to thank William Weihl, Chris Joerg, and John Dilley among many other Akamai engineers for their advice and suggestions during the design. We want to thank Gong Ke Shen for her role as a developer on this

project. We would like to thank Professor Jason Nieh for his motivation and advice with the paper. Finally, we want to thank all of the reviewers and especially our NSDI shepherd Jeff Mogul for their valuable comments.

References

[1] Akamai Technologies, Inc., *http://www.akamai.com/.*

[2] Network Operations Command Center, *http://www.akamai.com/en/html/technology/nocc.html.*

[3] *http://www.ntp.org/.*

[4] D.Karger, E.Lehman, T.Leighton, M.Levine, D.Lewin and R.Panigrahy, *Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web.* In Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, pages 654-663 , 1997.

[5] M. Castro, M. B. Jones, A-M Kermarrec, A. Rowstron, M. Theimer, H. Wang, and A. Wolman, *An Evaluation of Scalable Application-level Multicast Built Using Peer-to-Peer Overlays*, in Proc. INFOCOM, 2003.

[6] Y. Chawathe, S. McCanne, and E. Brewer, *RMX: Reliable Multicast for Heterogeneous Networks*, Proc. of INFOCOM, March 2000, pp. 795–804.

[7] Y. H. Chu, S. G. Rao, and H. Zhang, *A case for end system multicast*, Proc. of ACM Sigmetrics, June 2000, pp. 1–12.

[8] S. B. Davidson, H. Garcia-Molina, and D. Skeen, *Consistency in partitioned networks*, ACM Comput. Surveys, 1985.

[9] S. E. Deering and D. R. Cheriton, *Host extensions for IP multicasting*, Technical Report RFC 1112, Network Working Group, August 1989.

[10] C. J. Fidge, *Timestamp in Message Passing Systems that Preserves Partial Ordering*, Proc. 11th Australian Computing Conf., 1988, pp. 56–66.

[11] D. K. Gifford, *Weighted Voting for Replicated Data*, Proceedings 7th ACM Symposium on Operating Systems, 1979.

[12] J. Gray, *Notes on database operating systems*, Operating Systems: An Advanced Course. pp. 394–481, 1978.

[13] IBM Corporation, *WebSphere MQ family*, *http://www-306.ibm.com/software/integration/mqfamily/.*

[14] J. Jannotti, D. K. Gifford, K. L. Johnson, F. Kaashoek, and J. W. O'Toole, *Overcast: Reliable Multicasting with an Overlay Network*, Proc. of OSDI, October 2000, pp. 197–212.

[15] A. Kumar, *Hierarchical quorum consensus: A new algorithm for managing replicated data*, IEEE Trans. Computers, 1991.

[16] L. Lamport, *The Part-time Parliament*, ACM Transactions in Computer Systems, May, 1998.

[17] M. Castro, B. Liskov, *Practical Byzantine Fault-Tolerance*, OSDI 1999.

[18] L. Lamport, R. Shostak, and M. Pease, *The Byzantine Generals Problem*, ACM Transactions on Programming Languages and Systems, July 1982.

[19] N. Lynch and A. Shvartsman, *RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks*, DISC, October 2002.

[20] M. Satyanarayanan, *Scalable, Secure, and Highly Available Distributed File Access*, IEEE Computer, May 1990.

[21] S. Saito, C. Karamanolis, M. Karlsson, M. Mahalingam, *Taming Aggressive Replication in the Pangaea Wide-area File System*, OSDI 2002.

[22] K. Peterson, M. Spreitzer, D. Terry, *Flexible Update Propagation for Weakly Consistent Replication*, SOSP, 1997.

[23] S. Paul, K. Sabnani, J. C. Lin, and S. Bhattacharyya, *Reliable Multicast Transport Protocol (RMTP)*, IEEE Journal on Selected Areas in Communications, April 1997.

[24] F. Mattern, *Virtual Time and Global States of Distributed Systems*, Proc. Parallel and Distributed Algorithms Conf., Elsevier Science, 1988.

[25] Microsoft Corporation, *Microsoft Message Queuing (MSMQ) Center*, *http://www.microsoft.com/windows2000/technologies/communications/msmq/default.asp.*

[26] S. Rantmasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, *A Scalable Content-Addressable Network*, Proc. of ACM SIGCOMM, August 2001.

[27] S. Ratnasamy, M. Handley, R. Karp and S. Shenker, *Application-level multicast using content-addressable networks*, Proc. of NGC, November 2001.

[28] A. Rowstron and P. Drischel, *Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems*, Proc of Middleware, November 2001.

[29] A. Rowstron, A. M. Kermarrec, M. Castro and P. Druschel, *Scribe: The design of a large-scale event notification infra, structure* in Proc of NGC, Nov 2001.

[30] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, *Chord: A scalable peer-to-peer lookup service for internet applications*, in Proc of ACM SIGCOMM, August 2001.

[31] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, B. Zhao, *OceanStore: an architecture for global-scale persistent storage*, ASPLOS 2000, November 2000.

[32] Sun Microsystems, *Java Message Service*, *http://java.sun.com/products/jms.*

[33] B. Zhao, J. Kubiatowicz and A. Joseph, *Tapestry: An infrastructure for fault-resilient wide-area location and routing*, U. C. Berkeley, Tech. Rep. April 2001.

[34] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz, *Bayeux: An Architecture for Scalable and Fault-Tolerant Wide-Area Data Dissemination*, Proc. of NOSSDAV, June 2001.

[35] *http://www.lcfg.org/.*

[36] *http://www2.novadigm.com/hpworld/.*

[37] Microsoft Windows Update *http://windowsupdate.microsoft.com.*

The Collective: A Cache-Based System Management Architecture

Ramesh Chandra Nickolai Zeldovich Constantine Sapuntzakis Monica S. Lam
Computer Science Department
Stanford University
Stanford, CA 94305
{rameshch, nickolai, csapuntz, lam}@cs.stanford.edu

Abstract

This paper presents the Collective, a system that delivers managed desktops to personal computer (PC) users. System administrators are responsible for the creation and maintenance of the desktop environments, or *virtual appliances*, which include the operating system and all installed applications. PCs run client software, called the *virtual appliance transceiver*, that caches and runs the latest copies of appliances locally and continuously backs up changes to user data to a network repository. This model provides the advantages of central management, such as better security and lower cost of management, while leveraging the cost-effectiveness of commodity PCs.

With a straightforward design, this model provides a comprehensive suite of important system functions including machine lockdown, system updates, error recovery, backups, and support for mobility. These functions are made available to all desktop environments that run on the x86 architecture, while remaining protected from the environments and their many vulnerabilities. The model is suitable for managing computers on a LAN, WAN with broadband, or even computers occasionally disconnected from the network like a laptop. Users can access their desktops from any Collective client; they can also carry a bootable drive that converts a PC into a client; finally, they can use a remote display client from a browser to access their desktop running on a remote server.

We have developed a prototype of the Collective system and have used it for almost a year. We have found the system helpful in simplifying the management of our desktops while imposing little performance overhead.

1 Introduction

With the transition from mainframe computing to personal computing, the administration of systems shifted from central to distributed management. With mainframes, professionals were responsible for creating and maintaining the single environment that all users accessed. With the advent of personal computing, users got to define their environment by installing any software that fit their fancy. Unfortunately, with this freedom also came the tedious, difficult task of system management: purchasing the equipment and software, installing the software, troubleshooting errors, performing upgrades and re-installing operating systems, performing backups, and finally recovering from problems caused by mistakes, viruses, worms and spyware.

Most users are not professionals and, as such, do not have the wherewithal to maintain systems. As a result, most personal computers are not backed up and not up to date with security patches, leaving users vulnerable to data loss and the Internet vulnerable to worms that can infect millions of computers in minutes [16]. In the home, the challenges we have outlined above lead to frustration; in the enterprise, the challenges cost money.

The difficulties in managing distributed PCs has prompted a revival in interest in thin-client computing, both academically [15, 8] and commercially [2]. Reminiscent of mainframe computing, computation is performed on computers centralized in the data center. On the user's desk is either a special-purpose remote display terminal or a general-purpose personal computer running remote display software. Unfortunately, this model has higher hardware costs and does not perform as well. Today, the cheapest thin clients are PCs without a hard disk, but unlike a stand-alone PCs, a thin client also needs a server that does the computation, increasing hardware costs. The service provider must provision enough computers to handle the peak load; users cannot improve their computing experience by going to the store and buying a faster computer. The challenge of managing multiple desktops remains, even if it is centralized in the data center. Finally, remote display, especially over slower links, cannot deliver the same interactivity as local applications.

1.1 Cache-based System Management

This paper presents a cache-based system management model that combines the advantages of centralized management while taking advantage of inexpensive PCs. Our model delivers instances of the same software environments to desktop computers automatically, thereby amortizing the cost of the management. This design trades off users' ability to customize their own environment in return for uniformity, scalability, better security and lower cost of management.

In our model, we separate the state in a computer into two parts: system state and user state. The system state consists of an operating system and all installed applications. We refer to the system state as an *appliance* to emphasize that only the administrator is allowed to modify the system function; thus, to the user the system state defines a fixed function, just like any appliance. Note that these appliances are *virtual appliances* because unlike real appliances, they do not come with dedicated hardware. User state consists of a user's profile, preferences, and data files.

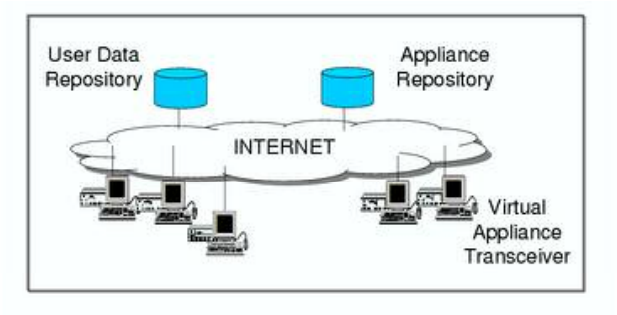


Figure 1: Architecture of the Collective system

In the cache-based system management model, appliances and user state are stored separately in network-accessible *appliance repositories* and *data repositories*, as shown in Figure 1. PCs in this model are fixed-function devices called *virtual appliance transceivers* (VATs).

In this model, a user can walk up to any of these clients, log in, and get access to any appliance he is entitled to. The VAT performs the following functions:

1. authenticates users.
2. fetches and runs the latest copies of appliances locally.
3. backs up user state changes to the data repository continuously; only administrators are allowed to update appliances in the appliance repository.
4. optimizes the system by managing a cache to reduce the amount of the data that needs to be fetched over the network.

Because appliances run locally, rather than on some central server, users experience performance and interactivity similar to their current PC environment; this approach provides the benefits of central management while leveraging commodity PCs.

1.2 System Highlights

We have developed a prototype based on this model which we call the Collective [14, 12]. By using x86 virtualization technology provided by the VMware GSX Server [20], the Collective client can manage and run most software that runs on an x86 PC. We have used the system daily since June 2004. Throughout this period, we have extended our system and rewritten parts of it several times. The system that we are describing is the culmination of many months of experience. This paper presents the design and rationale for cache-based system management, as well as the detailed design and implementation of the Collective. We also measure our prototype and describe our experiences with it.

Our cache-based system management model has several noteworthy characteristics. First, the VAT is a separate layer in the software stack devoted to management. The VAT is protected from the appliances it manages by the virtual machine monitor, increasing our confidence in the VAT's security and reliability in the face of appliance compromise or malfunction. Finally, the VAT automatically updates itself, requiring little to no management on the part of the user.

Second, the system delivers a comprehensive suite of critical system management functions automatically and efficiently, including disk imaging, machine lockdown, software updates, backups, system and data recovery, mobile computing, and disconnected operation, through a simple and unified design based on caching. Our cache design is kept simple with the use of a versioning scheme where every data item is referred to by a unique name. In contrast, these management functions are currently provided by a host of different software packages, often requiring manual intervention.

Third, our design presents a uniform user interface, while providing performance and security, across computers with different network connectivities and even on computers not running the VAT software. The design works on computers connected to a LAN or WAN with broadband bandwidth; it even works when computers are occasionally disconnected. It also enables a new mobility model, where users carry a portable storage device such as an 1.8-inch disk. With the disk, they can boot any compatible PC and run the VAT. Finally, in the case where users can only get access to a conventional PC, they can access their environment using a browser, albeit only at remote access speeds.

In contrast, without a solution that delivers performance across different network connectivities, enterprises often resort to using a complicated set of techniques as a compromise. In a large organization, users in the main campus may use PCs since they have enough IT staff to manage them; remote offices may use thin clients instead, trading performance for reduced IT staff. Employees may manage their home PCs, installing corporate applications, and accessing corporate infrastructure via VPN. Finally, ad hoc solutions may be used to manage laptops, as they are seldom connected to the network unless they are in use.

The combination of centralized management and cheap PCs, provided by the Collective, offers a number of practical benefits. This approach lowers the management cost at the cost of a small performance overhead. More importantly, this approach improves security by keeping software up to date and locking down desktops. Recovery from failures, errors, and attacks is made possible by continuous backup. And, the user sees the same computing environment at the office, at home, or on the road.

1.3 Paper Organization

The rest of the paper is structured as follows. Section 2 presents an overview of the system. Section 3 discusses the design in more detail. We present quantitative evaluation in section 4 and qualitative user experience in Section 5. Section 6 discusses the related work and Section 7 concludes the paper.

2 System Overview

This section provides an overview of the Collective. We start by presenting the data types in the system: appliances, repositories and subscriptions. We then show how the Collective works from a user's perspective. We describe how the Collective's architecture provides mobility and management functions and present optimizations that allow the Collective to perform well under different connectivities.

2.1 Appliances

An appliance encapsulates a computer state into a virtual machine which consists of the following:

- *System disks* are created by administrators and hold the appliance's operating system and applications. As part of the appliance model, the contents of the system disk at every boot are made identical to the contents published by the administrator. As the appliance runs, it may mutate the disk.
- *User disks* hold persistent data private to a user, such as user files and settings.
- *Ephemeral disks* hold user data that is not backed up. They are used to hold ephemeral data such browser caches and temporary files; there is little reason to incur additional traffic to back up such data over the network.
- A *memory image* holds the state of a suspended appliance.

2.2 Repositories

Many of the management benefits of the Collective derive from the versioning of appliances in network repositories. In the Collective, each appliance has a repository; updates to that appliance get published as a new version in that repository. This allows the VAT to automatically find and retrieve the latest version of the appliance.

To keep consistency simple, versions are immutable. To save space and to optimize data transfer, we use copy-on-write (COW) disks to express the differences between versions.

2.3 Subscriptions

Users have accounts, which are used to perform access control and keep per-user state. Associated with a user's account is the user's Collective profile which exists on network storage. In the user's Collective profile is a list of appliances that the user has access to. When the user first runs an appliance, a *subscription* is created in the profile to store the user state associated with that appliance.

To version user disks, each subscription in the user's Collective profile contains a repository for the user disks associated with the appliance. The first version in the repository is a copy of an initial user disk published in the appliance repository.

Other state associated with the subscription is stored only on storage local to the VAT. When an appliance starts, a COW copy of the system disk is created locally. Also, an ephemeral disk is instantiated, if the appliance requires one but it does not already exist. When an appliance is suspended, a memory image is written to the VAT's storage.

Since we do not transfer state between VATs directly, we cannot migrate suspended appliances between VATs. This is mitigated by the user's ability to carry their VAT with them on a portable storage device.

To prevent the complexity of branching the user disk, the user should not start a subscribed appliance on a VAT while it is running on another VAT. So that we can, in many cases, detect this case and warn the user, a VAT will attempt to acquire and hold a lock for the subscription while the appliance is running.

2.4 User Interface

The VAT's user interface is very simple–it authenticates the user and allows him to perform a handful of operations on appliances. On bootup, the VAT presents a Collective log-in window, where a user can enter his username and password. The system then presents the user with a list of appliances that he has access to, along with their status. Choosing from a menu, the user can perform any of the operations on his appliances:

- *start* boots the latest version of the appliance, or if a suspended memory image is available locally, resumes the appliance.
- *stop* shuts down the appliance.
- *suspend* suspends the appliance to a memory image.

- *reset* destroys all ephemeral disks and the memory image but retains the user disks
- *delete* destroys the subscription including the user disks
- *user disk undo* allows the user to go back to a previous snapshot of their user disk.
- *publish* allows the administrator to save the current version of the system disk as the latest version of the appliance.

When a user starts an appliance, the appliance takes over the whole screen once it runs; at any time, the user can hit a hot key sequence (Ctrl-Alt-Shift) and return to the list of appliances to perform other operations. The VAT user interface also indicates the amount of data that remains to be backed up from the local storage device. When this hits zero, the user can log out of the VAT and safely shift to using the virtual machines on another device.

2.5 Management Functions

We now discuss how the various management functions are implemented using caching and versioning.

2.5.1 System Updates

Desktop PCs need to be updated constantly. These upgrades include security patches to operating systems or installed applications, installations of new software, upgrades of the operating system to a new version, and finally re-installion of the operating system from scratch. All software upgrades, be they small or big, are accomplished in our system with the same mechanism. The system administrator prepares a new version of the appliance and deposits it in the appliance repository. The user gets the latest copy of the system disks the next time they reboot the appliance. The VAT can inform the user that a new version of the system disk is available, encouraging the user to reboot, or the VAT can even force a reboot to disallow use of the older version. From a user's standpoint, upgrade involves minimal work; they just reboot their appliances. Many software updates and installations on Windows already require the computer to be rebooted to take effect.

This update approach has some advantages over package and patch systems like yum [24], RPM [1], Windows Installer [22], and Windows Update. First, patches may fail on some users' computers because of interactions with user-installed software. Our updates are guaranteed to move the appliance to a new consistent state. Users running older versions are unaffected until they reboot. Even computers that have been off or disconnected get the latest software updates when restarted; with many patch management deployments, this is not the case. Finally, our update approach works no matter how badly the software in the appliance is functioning, since the VAT is protected from the appliance software. However, our model requires the user to subscribe to an entire appliance; patching works with individual applications and integrates well in environments where users mix and match their applications.

Fully automating the update process, the VAT itself is managed as an appliance. It automatically updates itself from images hosted in a repository. This is described in more detail in Section 3. By making software updates easy, we expect environments to be much more up to date with the Collective and hence more secure.

2.5.2 Machine Lockdown

Our scheme locks down user desktops because changes made to the system disks are saved in a new version of the disk and are discarded when the appliance is shut down. This means that if a user accidentally installs undesirable software like spyware into the system state, these changes are wiped out.

Of course, undesirable software may still install itself into the user's state. Even in this case, the Collective architecture provides the advantage of being able to reboot to an uncompromised system image with uncompromised virus scanning tools. If run before accessing ephemeral and user data, the uncompromised virus scanner can stay uncompromised and hopefully clean the changed state.

2.5.3 Backup

The VAT creates a COW snapshot of the user disk whenever the appliance is rebooted and also periodically as the appliance runs. The VAT backs up the COW disks for each version to the user repository. The VAT interface allows users to roll back changes made since the last reboot or to return to other previous versions. This allows the user to recover from errors, no matter the cause, be it spyware or user error.

When the user uses multiple VATs to access his appliances without waiting for backup to complete, potential for conflicts in the user disk repository arises. The backup protocol ensures that only one VAT can upload user disk snapshots into the repository at a time. If multiple VATs attempt to upload user disks at the same time, the user is first asked to choose which VAT gets to back up into the subscription and then given the choice of terminating the other backups or creating additional subscriptions for them.

If an appliance is writing and overwriting large quantities of data, and there is insufficient network bandwidth for backup, snapshots can accumulate at the client, buffered for upload. In the extreme, they can potentially fill the disk. We contemplate two strategies to deal with accumulating snapshots: collapse multiple snapshots to-

gether and reduce the frequency of snapshots. In the worst case, the VAT can stop creating new snapshots and collapse all of the snapshots into one; the amount of disk space taken by the snapshot is then bounded by the size of the virtual disk.

2.5.4 Hardware Management

Hardware management becomes simpler in the Collective because PCs running the VAT are interchangeable; there is no state on a PC that cannot be discarded. Deploying hardware involves loading PCs with the VAT software. Provisioning is easy because users can get access to their environments on any of the VATs. Faulty computers can be replaced without manually customizing the new computer.

2.6 Optimizations for Different Network Connectivities

To reduce network and server usage and improve performance, the VAT includes a large on-disk cache, on the order of gigabytes. The cache keeps local copies of the system and user disk blocks from the appliance and data repositories, respectively. Besides fetching data on demand, our system also prefetches data into the cache. To ensure good write performance even on low-bandwidth connections, all appliance writes go to the VAT's local disk; the backup process described in section 2.5.3 sends updates back in the background.

The cache makes it possible for this model to perform well under different network connectivities. We shall show below how our system allows users to access their environments on any computer, even computers with no pre-installed VAT client software, albeit with reduced performance.

2.6.1 LAN

On low-latency, high bandwidth (e.g., 100 Mbps) networks, the system performs reasonably well even if the cache is empty. Data can be fetched from the repositories fast enough to sustain good responsiveness. The cache is still valuable for reducing network and server bandwidth requirements. The user can easily move about in a LAN environment, since it is relatively fast to fetch data from a repository.

2.6.2 WAN with Broadband

By keeping a local copy of data from the repository, the cache reduces the need for data accesses over the network. This is significant because demand-fetching every block at broadband bandwidth and latency would make the system noticeably sluggish to the user. We avoid this worst-case scenario with the following techniques. First, at the time the VAT client software is installed on the hard disk, we also populate the cache with blocks of the appliances most likely to be used. This way only updates need to be fetched. Second, the VAT prefetches data in the background whenever updates for the appliances in use are available. If, despite these optimizations, the user wishes to access an appliance that has not been cached, the user will find the application sluggish when using a feature for the first time. The performance of the feature should subsequently improve as its associated code and data get cached. In this case, although the system may try the user's patience, the system is guaranteed to work without the user knowing details about installing software and other system administration information.

2.6.3 Disconnected Operation with Laptops

A user can ensure access to a warm cache of their appliances by carrying the VAT on a laptop. The challenge here is that a laptop is disconnected from the network from time to time. By hoarding all the blocks of the appliances the user wishes to use, we can keep operating even while disconnected.

2.6.4 Portable VATs

The self-containment of the VAT makes possible a new model of mobility. Instead of carrying a laptop, we can carry the VAT, with a personalized cache, on a bootable, portable storage device. Portable storage devices are fast, light, cheap, and small. In particular, we can buy a 1.8-inch, 40GB, 4200 rpm portable disk, weighing about 2 ounces, for about $140 today. Modern PCs can boot from a portable drive connected via USB.

The portable VAT has these advantages:

1. *Universality and independence of the computer hosts*. Eliminating dependences on the software of the hosting computer, the device allows us to convert any x86 PC into a Collective VAT. This approach leaves the hosting computer undisturbed, which is a significant benefit to the hosting party. Friends and relatives need not worry about their visitors modifying their computing environments accidentally, although malicious visitors can still wreak havoc on the disks in the computers.

2. *Performance*. The cache in the portable VAT serves as a network accelerator. This is especially important if we wish to use computers on low-bandwidth networks.

3. *Fault tolerance*. Under typical operation, the VAT does not contain any indispensable state when not in use; thus, in the event the portable drive is lost or forgotten, the user gets access to his data by inserting another generic VAT and continuing to work, albeit at a slower speed.

4. *Security and privacy*. This approach does not disturb the hosting computer nor does it leave any trace

of its execution on the hosting computer. Data on the portable drive can be encrypted to maintain secrecy if the portable drive is lost or stolen. However, there is always the possibility that the firmware of the computer has been doctored to spy on the computations being performed. Trusted computing techniques [18, 5] can be applied here to provide more security; hardware could in theory attest to the drive the identity of the firmware.

2.6.5 Remote Display

Finally, in case the users do not have access to any VATs, they can access their environments using remote display. We recreate a user experience similar to the one with the VAT; the user logs in, is presented with a list of appliances and can click on them to begin using them. The appliances are run on a server and a window appears with an embedded Java remote display applet that communicates with the server.

3 Design of the VAT

In this section, we present the design of the appliance transceiver. The VAT's major challenges include running on as many computers as possible, automatically updating itself, authenticating users and running their appliances, and working well on slow networks.

3.1 Hardware Abstraction

We would like the VAT image to run on as many different hardware configurations as possible. This allows users with a bootable USB drive to access their state from almost any computer that they might have available to them. It also reduces the number of VAT images we need to maintain, simplifying our administration burden.

To build a VAT that would support a wide range of hardware, we modified KNOPPIX [6], a *Live CD* version of Linux that automatically detects available hardware at boot time and loads the appropriate Linux drivers. KNOPPIX includes most of the drivers available for Linux today.

KNOPPIX's ability to quickly auto-configure itself to a computer's hardware allows the same VAT software to be used on many computers without any per-computer modification or configuration, greatly simplifying the management of an environment with diverse hardware. We have found only one common situation where the VAT cannot configure itself without the user's help: to join a wireless network, the user may need to select a network and provide an encryption key.

If a VAT successfully runs on a computer, we can be reasonably sure the appliances running on it will. The appliances run by the VAT see a reasonably uniform set of virtual devices; VMware GSX server takes advantage of the VAT's device drivers to map these virtual devices to a wide range of real hardware devices.

3.2 User Authentication

To maintain security, users must identify themselves to the VAT and provide credentials that will be used by the VAT to access their storage on their behalf. As part of logging in, the user enters his username and password. The VAT then uses SSH and the password to authenticate the user to the server storing the user's Collective profile. To minimize the lifetime of the user's password in memory, the VAT sets up a key pair with the storage server so that it can use a private key, rather than a password, to access storage on behalf of the user.

Disconnected operation poses a challenge, as there is no server to contact. However, if a user has already logged in previously, the VAT can authenticate him. When first created, the private key mentioned in the previous paragraph is stored encrypted with the user's password. On a subsequent login, if the password entered successfully decrypts the private key, the user is allowed to login and access the cached appliances and data.

3.3 VAT Maintenance

As mentioned earlier, the VAT is managed as an appliance, and needs zero maintenance from the end user; it automatically updates itself from a repository managed by a VAT administrator.

For most problems with the VAT software, a reboot restores the software to a working state. This is because the VAT software consists largely of a read-only file system image. Any changes made during a session are captured in separate file systems on a ramdisk. As a result, the VAT software does not drift from the published image.

All VATs run an update process which checks a repository for new versions of the VAT software and downloads them to the VAT disk when they become available. To ensure the integrity and authenticity of the VAT software, each version is signed by the VAT publisher. The repository location and the public key of publisher are stored on the VAT disk.

After downloading the updated image, the update process verifies the signature and atomically changes the boot sector to point to the new version. To guarantee progress, we allocate enough room for three versions of the VAT image: the currently running version, a potentially newer version that is pointed to by the boot sector which will be used at next reboot, and an even newer, incomplete version that is in the process of being downloaded or verified.

The VAT image is about 350MB uncompressed; downloading a completely new image wastes precious network capacity on broadband links. To reduce the size to about 160MB, we use the `cloop` tools that come with KNOPPIX to generate a compressed disk image; by using the `cloop` kernel driver, the VAT can mount the root file system from the compressed file system directly. Even so, we expect most updates to touch only a few files; transferring

an entire 160MB image is inefficient. To avoid transferring blocks already at the client, the update process uses rsync; for small changes, the size of the update is reduced from 160MB to about 10MB.

3.4 Storage Access

Our network repositories have the simplest layout we could imagine; we hope this will let us use a variety of access protocols. Each repository is a directory; each version is a subdirectory whose name is the version number. The versioned objects are stored as files in the subdirectories. Versions are given whole numbers starting at 1. Since some protocols (like HTTP) have no standard directory format, we keep a `latest` file in the repository's main directory that indicates the highest version number.

To keep consistency simple, we do not allow a file to be changed once it has been published into a repository. However, it should be possible to reclaim space of versions that are old; as such, files and versions can be deleted from the repository. Nothing prevents the deletion of an active version; the repository does not keep track of the active users.

When reading from network storage, we wanted a simple, efficient protocol that could support demand paging of large objects, like disk images. For our prototype, we use NFS. NFS has fast, reliable servers and clients. To work around NFS's poor authentication, we tunnel NFS over SSH.

While demand paging a disk from the repository, we may become disconnected. If a request takes too long to return, the disk drivers in the appliance's OS will timeout and return an I/O error. This can lead to file system errors which can cause the system to panic. In some cases, suspending the OS before the timeout and resuming it once we have the block can prevent these errors. A better solution is to try to make sure this does not happen in the first place by aggressively caching blocks; we will discuss this approach more in Section 3.6.

We also use NFS to write to network storage. To atomically add a new version to a repository, the VAT first creates and populates the new version under a one-time directory name. As part of the process, it places a nonce in the directory. The VAT then renames the directory to its final name and checks the nonce to see if it succeeded.

We would also like to set the priority of the writes to user data repositories with respect to other network traffic. Currently, the VAT approximates this by mounting the NFS server again on a different mount point; this in turn uses a separate TCP connection, which is given a different priority. Another consideration is that when an NFS server is slow or disconnected, the NFS in-kernel client will buffer writes in memory, eventually filling memory with dirty blocks and degrading performance. To limit the quantity of dirty data, the VAT performs an fsync after every 64 kilobytes of writes to the user data repository.

3.5 Caching

Our cache is designed to mask the high latency and low bandwidth of wide-area communication by taking advantage of large, persistent local storage, like hard disks and flash drives. At the extreme, the cache allows the client to operate disconnected.

COW disks can be gigabytes in size; whole file caching them would lead to impractical startup times. On the other hand, most of the other data in our system, like the virtual machine description, is well under 25 kilobytes. As a result, we found it easiest to engineer two caches: a small object cache for small data and meta-data and a COW cache for COW disk blocks.

To simplify disconnected operation, small objects, like the user's list of appliances or the meta-data associated with a repository, are replicated in their entirety. All the code reads the replicated copy directly; a background process periodically polls the servers for updates and integrates them into the local replica. User data snapshots, which are not necessarily small, are also stored in their entirety before being uploaded to the server.

The COW cache is designed to cache immutable versions of disks from repositories; as such the name of a disk in the cache includes an ID identifying the repository (currently the URL), the disk ID (currently the disk file name), and the version number. To name a specific block on a disk, the offset on disk is added. Since a data block can change location when COW disk chains are collapsed, we use the offset in the virtual disk, not the offset in the COW disk file.

One of the challenges of on-disk data structures is dealing with crashes, which can happen at any time, leading to partial writes and random data in files. With a large cache, scanning the disk after a crash is unattractive. To cope with partial writes and other errors introduced by the file system, each 512-byte sector stored in the cache is protected by an MD5 hash over its content and its address. If the hash fails, the cache assumes the data is not resident in the cache.

A traditional challenge with file caches has been invalidation; however, our cache needs no invalidation protocol. The names used when storing and retrieving data from the cache include the version number; since any given version of an object is immutable, no invalidation is necessary.

Our cache implementation does not currently make an effort to place sequential blocks close to each other on disk. As a result, workloads that are optimized for sequential disk access perform noticeably slower with our cache, due to the large number of incurred seeks. One such common workload is system bootup; we have implemented a *bootup block optimization* for this case. Since the block

access pattern during system bootup is highly predictable, a trace of the accessed blocks is saved along with each virtual appliance. When an appliance is started, the trace is replayed, bringing the blocks into the buffer cache before the appliance OS requests them. This optimization significantly reduces bootup time. A more general version of this technique can be applied to other predictable block access patterns, such as those associated with starting large applications.

3.6 Prefetching

To minimize cache misses, the VAT runs a prefetcher process to fetch useful appliance blocks in the background. The prefetcher checks for updates to appliances used by the VAT user, and populates the cache with blocks from the updated appliances. One optimization is to prioritize the blocks using access frequency so that the more important data can be prefetched first.

The user can use an appliance in disconnected mode by completely prefetching a version of an appliance into the cache. The VAT user interface indicates to the user what versions of his appliances have been completely prefetched. The user can also manually issue a command to the prefetcher if he explicitly wants to save a complete version of the appliance in the cache.

The prefetcher reduces interference with other processes by rate-limiting itself. It maintains the latencies of recent requests and uses these to determine the extent of contention for network or disk resources. The prefetcher halves its rate if the percentage of recent requests experiencing a high latency exceeds a threshold; otherwise, it doubles its rate when it finds a large percentage experience a low latency. If none of these apply, it increases or decreases request rate by a small constant based on the latency of the last request.

Prefetching puts spare resources to good use by utilizing them to provide better user experience in the future: when a user accesses an appliance version for the first time, it is likely that the relevant blocks would already be cached. Prefetching hides network latency from the appliance, and better utilizes network bandwidth by streaming data rather than fetching it on demand.

4 Evaluation

We provide some quantitative measurements of the system to give a sense of how the system behaves. We perform four sets of experiments. We first use a set of benchmarks to characterize the overhead of the system and the effect of using different portable drives. We then present some statistics on how three appliances we have created have evolved over time. Next, we evaluate prefetching. We show that a small amount of data accounts for most of the accesses, and that prefetching can greatly improve the responsiveness of an interactive workload. Finally, we study the feasibility of continuous backups.

4.1 Run-Time Performance

We first establish some basic parameters of our system by running a number of benchmarks under different conditions. All of the experiments, unless noted otherwise, are run on 2.4GHz Pentium IV machines with 1GB of memory and a 40GB Hitachi 1.8" hard drive connected via Prolific Technology's PL-2507 USB-to-IDE bridge controller. VAT software running on the experimental machines is based on Linux kernel 2.6.11.4 and VMware GSX server version 3.1. The file server is a 2.4GHz Pentium IV with 1GB of memory and a Linux software RAID, consisting of four 160GB IDE drives. We use FreeBSD's *dummynet* [11] network simulator to compare the performance of our system over a 100 Mbps LAN to that over a 1.5 Mbps downlink / 384 Kbps uplink DSL connection with 40 msec round-trip delay.

4.1.1 Effects of Caching

To evaluate caching, we use three repeatable workloads: bootup and shutdown of a Linux VM, bootup and shutdown of a Windows XP VM, and building the Linux 2.4.23 kernel in a VM. The runtime of each workload is measured in different network and cache configurations to illustrate how caching and network connectivity affect performance. All workloads are run with both an empty and a fully prefetched initial cache. We also repeat the workloads with a fully prefetched cache but without the bootup block optimization, to show the optimization's effect on startup performance.

By running the same virtual machine workloads on an unmodified version of VMware's GSX server, we quantify the benefits and overheads imposed by the Collective caching system. In particular, we run two sets of experiments using unmodified VMware without the Collective cache:

- *Local*, where the entire VM is copied to local disk and executes without demand-fetching. The COW disks of each VM disk are collapsed into a flat disk for this experiment. We expect this to provide a bound on VMware performance.

- *NFS*, where the VM is stored on an NFS file server and is demand-fetched by VMware without additional caching. This is expected to be slow in the DSL case and shows the need for caching.

Figure 2 summarizes the performance of these benchmarks. Workloads running with a fully prefetched cache are slower than the *local* workload, due to additional seek overhead imposed by the layout of blocks in our cache. The bootup block prefetching optimization, described in

Section 3.5, largely compensates for the suboptimal block layout.

As expected, the performance of our workloads is bad in both the NFS and the empty cache scenario, especially in the case of a DSL network, thus underscoring the need for caching.

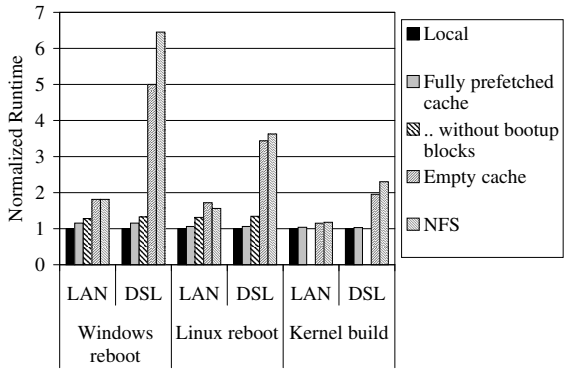


Figure 2: Runtime of workload experiments on different cache configurations when run over a 100 Mbps LAN, and a simulated DSL link with 1.5 Mbps downlink / 384 Kbps uplink and 40 msec RTT latency. The runtimes are normalized to the runtime in the local experiment. The local runtimes are 64 sec, 32 sec, and 438 sec, respectively for the Windows reboot, Linux reboot, and Linux kernel build experiments.

4.1.2 Effects of disk performance

As a first test to evaluate the performance of different disks, we measured the time taken to boot the VAT software on an IBM Thinkpad T42p laptop, since our standard experimental desktop machine did not have USB 2.0 support in its BIOS. The results, shown in the first column of Figure 3 indicate that the VAT boot process is reasonably fast across different types of drives we tried.

For our second test, we run the same micro-benchmark workloads as above; to emphasize disk performance rather than network performance, the VMs are fully prefetched into the cache, and the machines are connected over a 100Mbps LAN. The results are shown in Figure 3. The flash drive performs well on this workload, because of its good read performance with zero seek time, but has limited capacity, which would prevent it from running larger applications well. The microdrive is relatively slow, largely due to its high seek time and rotational latency. In our opinion, the 1.8" hard drive offers the best price / performance / form factor combination.

4.2 Maintaining Appliances

We have created and maintained three virtual machine appliances over a period of time:

- a Windows XP environment. Over the course of half a year, the Windows appliance has gone through two service packs and many security updates. The appliance initially contained Office 2000 and was upgraded to Office 2003. The appliance includes a large number of applications such as Adobe PhotoShop, FrameMaker, and Macromedia DreamWeaver.

- a Linux environment, based on Red Hat's Fedora Core, that uses NFS to access our home directories on our group file server. Over a period of eight months, the NFS Linux appliance required many security updates, which replaced major subsystems like the kernel and X server. Software was added to the NFS Linux appliance as it was found to be needed.

- a Linux environment also based on Fedora, that stores the user's home directory in a user disk. This Linux appliance included all the programs that came with the distribution and was therefore much larger. We used this appliance for two months.

	VAT startup	Windows reboot	Linux reboot	Kernel build
Lexar 1GB Flash Drive	53	129	42	455
IBM 4GB Microdrive	65	158	53	523
Hitachi 40GB 1.8" Drive	61	84	43	457
Fujitsu 60GB 2.5" Drive	52	65	40	446

Figure 3: Performance characteristics of four different VAT disks. All the numbers are in seconds. The first column shows the VAT boot times on an IBM Thinkpad T42p, from the time BIOS transfers control to the VAT's boot block to the VAT being fully up and running. In all cases the BIOS takes an additional 8 seconds initializing the system before transferring control to the VAT. The rest of the table shows results for the micro-benchmarks run with fully-primed caches when run over a 100 Mbps network.

Some vital statistics of these appliances are shown in Figure 4. We show the number of versions created, either due to software installations or security patches. Changes to the system happen frequently; we saved a lot of time by having to just update one instance of each appliance.

Appliance	Number of versions	Total size	Active size	Cache size
Windows XP	31	16.5	4.5	3.1
NFS Linux	20	5.7	2.8	1.4
User-disk Linux	8	7.0	4.9	3.7

Figure 4: Statistics of three appliances. Sizes are in GB.

We also measure the size of all the COW disks for each appliance ("Total size") and the size of the latest version ("Active size"). The last column of the table, "Cache size", shows an example of the cache size of an active user of each appliance. We observe from our usage that the cache size grows quickly and stabilizes within a short amount of time. It grows whenever major system updates are performed and when new applications are used for the first time. The sizes shown here represent all the blocks ever cached and may include disk blocks that may have

since been made obsolete. We have not needed to evict any blocks from our 40GB disks.

4.3 Effectiveness of Prefetching

In the following, we first measure the access profile to establish that prefetching a small amount of data is useful. Second, we measure the effect of prefetching on the performance of an interactive application.

4.3.1 Access Profile

In this experiment, we measure the access profile of appliance blocks, to understand the effectiveness of prefetching based on the popularity of blocks. We took 15 days of usage traces from 9 users using the three appliances described above in their daily work. Note that during this period some of the appliances were updated, so the total size of data accessed was greater than the size of a single active version. For example, the Windows XP appliance had an active size of 4.5 GB and seven updates of 4.4 GB combined, for a total of 8.9 GB of accessible appliance data.

Figure 5 shows each appliance's effective size, the size of all the accesses to the appliance in the trace, and the size of unique accesses. The results suggest that only a fraction of the appliance data is ever accessed by any user. In this trace, users access only 10 to 30% of the accessible data in the appliances.

Appliance	Accessible Size	Accesses in Traces	Unique data Accessed
Windows XP	8.9 GB	31.1 GB	2.4 GB
NFS Linux	3.4 GB	6.8 GB	1.0 GB
User-disk Linux	6 GB	5.9 GB	0.5 GB

Figure 5: Statistics of appliances in the trace.

Figure 6 shows the percentage of accesses that are satisfied by the cache (Y-axis) if a given percentage of the most popular blocks are cached (X-axis). The results show that a large fraction of data accesses are to a small fraction of the data. For example, more than 75% of data accesses in the Windows XP appliance are to less than 20% of the accessed data, which is about 5% of the total appliance size. These preliminary results suggest that popularity of accessed appliance data is a good heuristic for prefetching, and that prefetching a small fraction of the appliance's data can significantly reduce the chances of a cache miss.

4.3.2 Interactive Performance

The responsiveness of an interactive application can be severely affected by cache miss delays. Our next experiment attempts to measure the effects of prefetching on an application's response time.

To simulate interactive workloads, we created a VNC [10] *recorder* to record user mouse and keyboard input events, and a VNC *player* to play them back to reproduce user's actions [25]. Using VNC provides us with

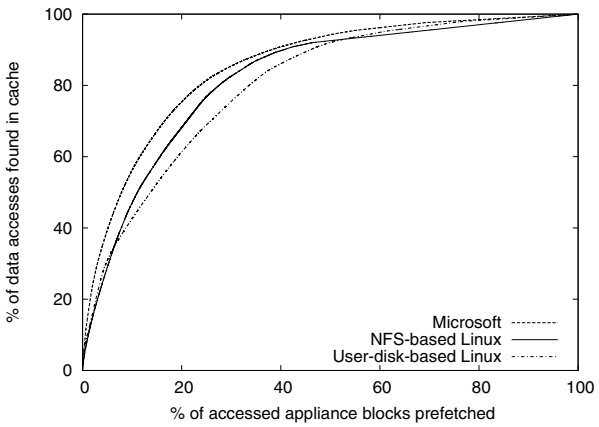


Figure 6: Block access profile: cache hit rate as a function of prefetched appliance data. Most frequently used appliance data is prefetched first.

a platform-independent mechanism for interacting with the desktop environment. Furthermore, it allows us to use VMware's built-in VNC interface to the virtual machine console.

Other tools [19, 23] try to do this, but play back is not always correct when the system is running significantly slower (or faster) than during recording. This is especially true for mouse click events. To reliably replay user actions, our VNC recorder takes screen snapshots along with mouse click events. When replaying input events, the VNC player waits for the screen snapshot taken during recording to match the screen contents during replay before sending the mouse click.

Our replay works only on systems with little or no nondeterministic behavior. Since we use virtual machines, we can easily ensure that the initial state is the same for each experiment.

We use the Windows XP appliance to record a VNC session of a user creating a PowerPoint presentation for approximately 8 minutes in a LAN environment. This session is then replayed in the following experimental configurations:

- Local: the entire appliance VM is copied to the VAT disk and executed with unmodified VMware, without demand-fetching or caching.
- Prefetched: some of the virtual machine's blocks are prefetched into the VAT's cache, and the VM is then executed on top of that cache. The VAT is placed behind a simulated 1.5 Mbps / 384 Kbps DSL connection.

For the prefetched experiments, we asked four users to use various programs in our appliance, to model other people's use of the same appliance; their block access patterns are used for prefetching blocks in the experiment.

Prefetching measures the amount of data transferred over the network; due to compression, the amount of raw disk data transferred is approximately 1.6 times more. The amount of prefetching goes up to a maximum of 420 MB, which includes all of the blocks accessed in the appliance by our users.

The total runtimes for the replayed sessions are within approximately 5% of each other – the additional latency imposed by demand-fetching disk blocks over DSL is absorbed by long periods of user think time when the system is otherwise idle. To make a meaningful comparison of the results, we measure the response time latency for each mouse click event, and plot the distribution of response times over the entire workload in Figure 7. For low response times, the curves are virtually indistinguishable. This region of the graph corresponds to events that do not result in any disk access, and hence are quick in all the scenarios. As response time increases, the curves diverge; this corresponds to events which involve accessing disk – the system takes noticeably longer to respond in this case, when disk blocks need to be demand-fetched over the network. The figure shows that PowerPoint running in the Collective is as responsive as running in a local VM, except for times when new features have to be loaded from disk – similar to Windows taking a while to start any given application for the first time.

The most commonly accessed blocks are those used in the bootup process. This experiment only measures the time taken to complete the PowerPoint workload after the system has been booted up, and therefore the benefit of prefetching the startup blocks is not apparent in the results shown in the figure. However, prefetching the startup blocks (approximately 100 MB) improves startup time from 391 seconds in the no prefetching case to 127 seconds when 200 MB of data is prefetched.

The results show that prefetching improves interactive performance. In the case of full prefetching, the performance matches that of a local VM. Partial prefetching is also beneficial – we can see that prefetching 200 MB significantly improves the interactive performance of PowerPoint.

4.4 Feasibility of Online Backup

Ideally, in our system, user data should always be backed up onto network storage. To determine whether online backup works for real workloads, we collected usage traces for three weeks on personal computers of ten users running Windows XP. These users included office workers, home users, and graduate students. The traces contain information on disk block reads and writes, file opens and start and end of processes. We also monitored idle times of keyboard and mouse; we assume the user to be idle if the idle time exceeds five minutes.

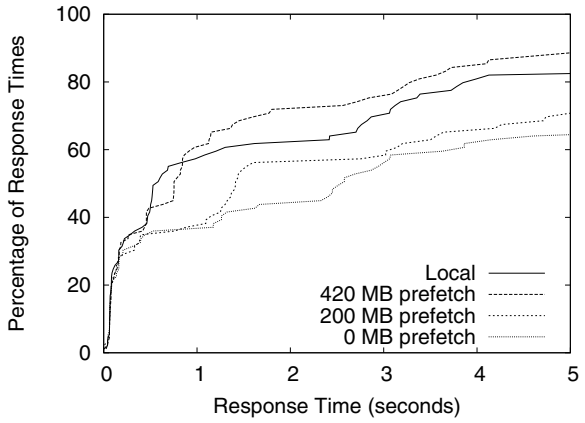


Figure 7: CDF plot of response times observed by the user during a PowerPoint session, for different levels of prefetching.

We expect that in our system the user would log out and possibly shut down his VAT soon after he completes his work. So, the measure we are interested in is whether there is any data that is not backed up when he becomes idle. If all the data is backed up, then the user can log in from any other VAT and get his most recent user data; if the user uses a portable VAT, he could lose it with no adverse effects.

To quantify this measure we simulated the usage traces on our cache running over a 384 Kbps DSL uplink. To perform the simulation we divided the disk writes from the usage data into writes to system data, user data, and ephemeral data. These correspond to the system disk, user disk, and ephemeral disk that were discussed earlier. System data consists of the writes that are done in the normal course by an operating system that need not be backed up. Examples of this include paging, defragmentation, NTFS metadata updates to system disk, and virus scans. User data consists of the data that the user would want to be backed up. This includes email documents, office documents, etc., We categorize internet browser cache, and media objects such as mp3 files, that are downloaded from the web as ephemeral data and do not consider them for backup. In our traces there were a total of about 300GB worth of writes of which about 3.3% were to user data, 3.4% were to ephemeral data and the rest to program data. Users were idle 1278 times in the trace, and in our simulation, backup stops during idle periods. We estimate the size of dirty data in the cache when users become idle.

The results are presented in Figure 8. The x-axis shows the size of data that is not backed up, and the y-axis shows the percentage of idle periods. From the figure we see that most of the time there is very little data to be backed up by the time the user becomes idle. This suggests that interactive users have large amounts of think time and generate little backup traffic. This also shows that online backup,

as implemented in the Collective, works well even on a DSL link. Even in the worst case, the size of dirty data is only about 35 MB, which takes less than 15 minutes to backup on DSL.

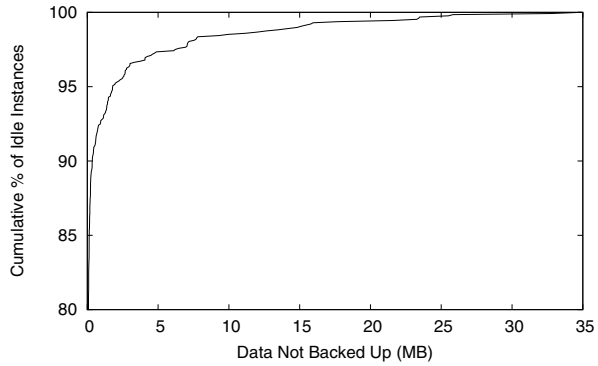


Figure 8: Size of data that is not backed up when a user becomes idle. The graph shows the fraction of times the user would have less than a certain amount of dirty data in his cache at the end of his session.

The results presented in this section illustrate that the system performs well over different network connections, and that it provides a good interactive user experience. Further, the results support our use of prefetching for reducing cache misses, and show that continuous backup is feasible for most users.

5 Experiences

We have been using the Collective for our daily work since June 2004. Based on this and other experiences, we describe how the Collective helped reduce the burden of administering software and computers.

5.1 Uses of the System

At first, members of our research group were using the prototype for the sake of understanding how our system behaves. As the system stabilized, more people started using the Collective because it worked better than their current setup. The following real-life scenarios we encountered illustrate some of the uses of our system:

Deploying new equipment. Before the Collective, when we needed to set up a new desktop or laptop, it would take a couple of hours to install the operating system, applications, and configure the computer. By plugging in and booting from USB disk containing the VAT, we were able to start using the computer immediately, starting up appliances we had previously used on other computers. We also used the VAT to configure the new computer's internal hard drive to be a VAT; all it takes is one user command and, in less than 5 minutes, the computer is assimilated into the Collective.

Fixing broken software setups. In one case, a student adopted the Collective after he botched the upgrade of the Linux kernel on his laptop. As a result of the failed upgrade, the laptop did not even boot. In other cases, we had lent machines to other groups and received them back with less than useful software setups. The Collective allowed us to resume work quickly by placing a VAT on the computer.

Distributing a complex computing environment. Over the summer, two undergraduates participated in a compiler research project that required many tools including Java, Eclipse, the JoeQ Java compiler, BDD libraries, etc. Since the students were not familiar with those tools, it would have taken each of the students a couple of days to create a working environment. Instead, an experienced graduate student created an appliance that he shared with both students, enabling both of them to start working on the research problems.

Using multiple environments. Our Linux appliance users concurrently start up a Windows appliance for the occasional tasks, like visiting certain web pages and running Powerpoint, that work better or require using Windows applications.

Distributing a centrally maintained infrastructure. Our university maintains a pool of computers that host the software for course assignments. Towards the end of the term, these computers become over-subscribed and slow. While the course software and the students' home directories are all available over a distributed file system (AFS), most students do not want to risk installing Linux and configuring AFS on their laptops. We gave students external USB drives with a VAT and created a Linux appliance that uses AFS to access course software and their home directories. The students used the VAT and the appliance to take advantage of the ample cycles on their laptops, while leaving the Windows setup on their internal drive untouched.

5.2 Lessons from our Experience

We appreciate that we only need to update an appliance once and all of the users can benefit from it. The authors would not be able to support all the users of the system otherwise.

The design of the VAT as a portable, self-contained, fixed-function device contributes greatly to our ability to carry out our experiments.

1. Auto-update. It is generally hard to conduct experiments involving distributed users because the software being tested needs to be fixed and improved frequently, especially at the beginning. Our system automatically updates itself allowing us to make quick iterations in the experiment without having to recall the experiment. The user needs to take no action, and the system has the appearance of healing itself upon a reboot.

2. Self-containment. It is easy to get users to try out the system because we give them an external USB drive from which to boot their computer. The VAT does not disturb the computing environment stored on their internal hard drive.

The system also makes us less wary of taking actions that may compromise an appliance. For example, we can now open email attachments more willingly because our system is up to date with security patches, and we can roll back the system should the email contain a new virus. As a trial, we opened up a message containing the BagleJ email virus in a system that had not yet been patched. Because BagleJ installed itself onto the system disk, it was removed when we rebooted. We have had similar experiences with spyware; a reboot removes the spyware executables, leaving only some icons on the user's desktop to clean up.

We observed that the system can be slow when it is used to access appliance versions that have not yet been cached. This is especially true over a DSL network. Prefetching can be useful in these cases. Prefetching on a LAN is fast; on a DSL network, it is useful to leave the computer connected to the network even when it is not in use, to allow prefetching to complete. The important point to note here is that this is fully automatic and hands-free, and it is much better than having to baby-sit the software installation process. Our experience suggests that it is important to prioritize between the different kinds of network traffic performed on behalf of the users; background activities like prefetching new appliance versions or backing up user data snapshots should not interfere with normal user activity.

We found that the performance of the Collective is not satisfactory for I/O intensive applications such as software builds, and graphics intensive applications such as video games. The virtualization overhead, along with the I/O overhead of our cache makes the Collective not suitable for these applications.

Finally, many software licenses restrict the installation of software to a single computer. Software increasingly comes with activation and other copy protection measures. Being part of a large organization that negotiates volume licenses, we avoided these licensing issues. However, the current software licensing model will have to change for the Collective model to be widely adopted.

6 Related Work

To help manage software across wide-area grids, GVFS [26] transfers hardware-level virtual machines. Their independent design shares many similarities to our design, including on-disk caches, NFS over SSH, and VMM-specific cache coherence. The Collective evaluates a broader system, encompassing portable storage, user data, virtual appliance transceiver, and initial user experiences.

Internet Suspend/Resume (ISR) [7] uses virtual machines and a portable cache to provide mobility; the Collective architecture also provides management features like rollback and automatic update, in addition to mobility. Similar to our previous work [13], ISR uses a cache indexed by content hash. In contrast, the current Collective prototype uses COW disks and a cache indexed by location. We feel that any system like the Collective needs COW disks to succinctly express versions; also, indexing the cache by location was straightforward to implement. Index by hash does have the advantage of being able to use a cached block from an unrelated disk image. Our previous work [13] suggests that there is promise in combining COW disks and index by hash. In the case a user does not wish to carry portable storage, ISR also implements *proactive* prefetching, which sends updated blocks to the computers the user uses commonly in anticipation of the user arriving there. The Collective uses prefetching of data from repositories to improve the performance at VATs where the user is already logged in. The two approaches are complementary.

Managing software using disk images is common; a popular tool is Symantec Ghost [17]. Unlike our system, a compromised operating system can disable Ghost since the operating system has full access to the raw hardware. In addition, since Ghost does not play copy-on-write tricks, roll back involves rewriting the whole partition. This potentially lengthy process limits the frequency of ghosting. Finally, Ghost leaves it to the administrator and other tools to address how to manage user data.

Using network repositories for disk images and expressing updates compactly using differences are explored by Rauch et al [9]. A different way of distributing disk images is Live CDs, bootable CDs with a complete software environment. Live CDs provide lock down and can easily roll back changes to operating systems. However, they do not provide automatic updates and management of user data.

Various solutions for transparent install and update exist for platforms other than x86 hardware. Java has Java Web Start [21]; some Windows games use Valve Steam; Konvalo and Zero Install manage Linux applications. The Collective uses virtual machine technology and an automatically updating virtual appliance transceiver to manage the entire software stack.

Like the Collective, MIT's Project Athena [4] provides the management benefits of centralized computing while using the power of distributed desktop computers. In Athena, management is a service that runs alongside applications; in contrast, the Collective's management software are protected from the applications by a virtual machine monitor. The Collective uses a disk-based ab-

straction to distribute software and user data; in contrast, Athena uses a distributed file system. By explicitly exposing multiple versions of disk images through repositories, the Collective can provide consistent snapshots of software and does not force users to start using the new version immediately. In contrast, software run from a network file system must be carefully laid out and managed to provide similar semantics. In Athena, users can mix and match software from many providers; in our model, an appliance is a monolithic unit created and tested by an administrator.

Candea et al [3] have explored rebooting components of a running system as a simple, consistent, and fast method of recovery. The Collective uses reboots to rollback changes and provide upgrades, providing similar advantages.

7 Conclusions

This paper presents the Collective, a prototype of a system management architecture for managing desktop computers. This paper concentrates on the design issues of a complete system. By combining simple concepts of caching, separation of system and user state, network storage, and versioning, the Collective provides several management benefits, including centralized management, atomic updates, and recovery via rollback.

Our design of a portable, self-managing virtual appliance transceiver makes the Collective infrastructure itself easy to deploy and maintain. Caching in the Collective helps provide good interactive performance even over wide-area networks. Our experience and the experimental data gathered on the system suggest that the Collective system management architecture can provide a practical solution to the complex problem of system management.

8 Acknowledgments

The work for this paper was funded in part by the National Science Foundation under Grant No. 0121481. We would like to thank Ben Pfaff, Chris Unkel, Emre Kıcıman, George Candea, Arvind Arasu, Mendel Rosenblum, Eu-jin Goh, our shepherd Ed Lazowska, and the anonymous reviewers for their comments.

References

[1] E. Bailey. *Maximum RPM*. Sams, 1st edition, 1997.

[2] W. M. Bulkeley. The office PC slims down. *The Wall Street Journal*, January 2005.

[3] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – a technique for cheap recovery. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 31–44, December 2004.

[4] G. Champine, J. Daniel Geer, and W. Ruh. Project Athena as a distributed computer system. *IEEE Computer Magazine*, 23(9):40–51, September 1990.

[5] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th Symposium on Operating System Principles(SOSP 2003)*, pages 193–206, October 2003.

[6] KNOPPIX Live CD Linux distribution. `http://www.knoppix.org/`.

[7] M. Kozuch, M. Satyanarayanan, T. Bressoud, C. Helfrich, and S. Sinnamohideen. Seamless mobile computing on fixed infrastructure. Technical Report 28, Intel Research Pittsburgh, 2004.

[8] J. Nieh, S. J. Yang, and N. Novik. Measuring thin-client performance using slow-motion benchmarking. *ACM Transactions on Computer Systems*, 21(1), February 2003.

[9] F. Rauch, C. Kurmann, and T. Stricker. Partition repositories for partition cloning—OS independent software maintenance in large clusters of PCs. In *Proceedings of the IEEE International Conference on Cluster Computing 2000*, pages 233–242, November/December 2000.

[10] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, January/February 1998.

[11] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, January 1997.

[12] C. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, J. Norris, M. S. Lam, and M. Rosenblum. Virtual appliances for deploying and maintaining software. In *Proceedings of Seventeenth USENIX Large Installation System Administration Conference*, pages 181–194, October 2003.

[13] C. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 377–390, December 2002.

[14] C. Sapuntzakis and M. Lam. Virtual appliances in the collective: A road to hassle-free computing. In *Workshop on Hot Topics in Operating Systems*, pages 55–60, May 2003.

[15] B. K. Schmidt, M. S. Lam, and J. D. Northcutt. The interactive performance of SLIM: a stateless, thin-client architecture. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, pages 32–47, December 1999.

[16] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the Internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium*, pages 149–167, August 2002.

[17] Symantec Ghost. `http://www.ghost.com/`.

[18] TCPA. `http://www.trustedcomputing.org/`.

[19] Rational VisualTest. `http://www.ibm.com/software/awdtools/tester/robot/`.

[20] VMware GSX server. `http://www.vmware.com/products/server/gsx_features.html`.

[21] Java web start. `http://java.sun.com/j2se/1.5.0/docs/guide/javaws/`.

[22] P. Wilson. *Definitive Guide to Windows Installer*. Apress, 1st edition, 2004.

[23] Xnee home page. `http://www.gnu.org/software/xnee/www`.

[24] Yellowdog updater modified (yum). `http://linux.duke.edu/projects/yum/`.

[25] N. Zeldovich and R. Chandra. Interactive performance measurement with VNCplay. In *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference*, pages 153–162, April 2005.

[26] M. Zhao, J. Zhang, and R. Figueiredo. Distributed file system support for virtual machines in grid computing. In *Proceedings of the Thirteenth IEEE Symposium on High-Performance Distributed Computing*, June 2004.

Live Migration of Virtual Machines

Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen†,
Eric Jul†, Christian Limpach, Ian Pratt, Andrew Warfield
University of Cambridge Computer Laboratory
15 JJ Thomson Avenue, Cambridge, UK
firstname.lastname@cl.cam.ac.uk
† *Department of Computer Science*
University of Copenhagen, Denmark
{jacobg,eric}@diku.dk

Abstract

Migrating operating system instances across distinct physical hosts is a useful tool for administrators of data centers and clusters: It allows a clean separation between hardware and software, and facilitates fault management, load balancing, and low-level system maintenance.

By carrying out the majority of migration while OSes continue to run, we achieve impressive performance with minimal service downtimes; we demonstrate the migration of entire OS instances on a commodity cluster, recording service downtimes as low as $60ms$. We show that that our performance is sufficient to make live migration a practical tool even for servers running interactive loads.

In this paper we consider the design options for migrating OSes running services with liveness constraints, focusing on data center and cluster environments. We introduce and analyze the concept of *writable working set*, and present the design, implementation and evaluation of high-performance OS migration built on top of the Xen VMM.

1 Introduction

Operating system virtualization has attracted considerable interest in recent years, particularly from the data center and cluster computing communities. It has previously been shown [1] that paravirtualization allows many OS instances to run concurrently on a single physical machine with high performance, providing better use of physical resources and isolating individual OS instances.

In this paper we explore a further benefit allowed by virtualization: that of live OS migration. Migrating an entire OS and all of its applications as one unit allows us to avoid many of the difficulties faced by process-level migration approaches. In particular the narrow interface between a virtualized OS and the virtual machine monitor (VMM) makes it easy avoid the problem of 'residual dependencies' [2] in which the original host machine must remain available and network-accessible in order to service certain system calls or even memory accesses on behalf of migrated processes. With virtual machine migration, on the other hand, the original host may be decommissioned once migration has completed. This is particularly valuable when migration is occurring in order to allow maintenance of the original host.

Secondly, migrating at the level of an entire virtual machine means that in-memory state can be transferred in a consistent and (as will be shown) efficient fashion. This applies to kernel-internal state (e.g. the TCP control block for a currently active connection) as well as application-level state, even when this is shared between multiple cooperating processes. In practical terms, for example, this means that we can migrate an on-line game server or streaming media server without requiring clients to reconnect: something not possible with approaches which use application-level restart and layer 7 redirection.

Thirdly, live migration of virtual machines allows a separation of concerns between the users and operator of a data center or cluster. Users have 'carte blanche' regarding the software and services they run within their virtual machine, and need not provide the operator with any OS-level access at all (e.g. a root login to quiesce processes or I/O prior to migration). Similarly the operator need not be concerned with the details of what is occurring within the virtual machine; instead they can simply migrate the entire operating system and its attendant processes as a single unit.

Overall, live OS migration is a extremelely powerful tool for cluster administrators, allowing separation of hardware and software considerations, and consolidating clustered hardware into a single coherent management domain. If a physical machine needs to be removed from service an administrator may migrate OS instances including the applications that they are running to alternative machine(s), freeing the original machine for maintenance. Similarly, OS instances may be rearranged across machines in a cluster to relieve load on congested hosts. In these situations the combination of virtualization and migration significantly improves manageability.

We have implemented high-performance migration support for Xen [1], a freely available open source VMM for commodity hardware. Our design and implementation addresses the issues and tradeoffs involved in live local-area migration. Firstly, as we are targeting the migration of active OSes hosting live services, it is critically important to minimize the *downtime* during which services are entirely unavailable. Secondly, we must consider the *total migration time*, during which state on both machines is synchronized and which hence may affect reliability. Furthermore we must ensure that migration does not unnecessarily disrupt active services through resource contention (e.g., CPU, network bandwidth) with the migrating OS.

Our implementation addresses all of these concerns, allowing for example an OS running the SPECweb benchmark to migrate across two physical hosts with only $210ms$ unavailability, or an OS running a Quake 3 server to migrate with just $60ms$ downtime. Unlike application-level restart, we can maintain network connections and application state during this process, hence providing effectively seamless migration from a user's point of view.

We achieve this by using a *pre-copy* approach in which pages of memory are iteratively copied from the source machine to the destination host, all without ever stopping the execution of the virtual machine being migrated. Page-level protection hardware is used to ensure a consistent snapshot is transferred, and a rate-adaptive algorithm is used to control the impact of migration traffic on running services. The final phase pauses the virtual machine, copies any remaining pages to the destination, and resumes execution there. We eschew a 'pull' approach which faults in missing pages across the network since this adds a residual dependency of arbitrarily long duration, as well as providing in general rather poor performance.

Our current implementation does not address migration across the wide area, nor does it include support for migrating local block devices, since neither of these are required for our target problem space. However we discuss ways in which such support can be provided in Section 7.

2 Related Work

The Collective project [3] has previously explored VM migration as a tool to provide mobility to users who work on different physical hosts at different times, citing as an example the transfer of an OS instance to a home computer while a user drives home from work. Their work aims to optimize for slow (e.g., ADSL) links and longer time spans, and so stops OS execution for the duration of the transfer, with a set of enhancements to reduce the transmitted image size. In contrast, our efforts are concerned with the migration of live, in-service OS instances on fast neworks with only tens of milliseconds of downtime. Other projects that have explored migration over longer time spans by stopping and then transferring include Internet Suspend/Resume [4] and μDenali [5].

Zap [6] uses partial OS virtualization to allow the migration of process domains (pods), essentially process groups, using a modified Linux kernel. Their approach is to isolate all process-to-kernel interfaces, such as file handles and sockets, into a contained namespace that can be migrated. Their approach is considerably faster than results in the Collective work, largely due to the smaller units of migration. However, migration in their system is still on the order of seconds at best, and does not allow live migration; pods are entirely suspended, copied, and then resumed. Furthermore, they do not address the problem of maintaining open connections for existing services.

The live migration system presented here has considerable shared heritage with the previous work on NomadBIOS [7], a virtualization and migration system built on top of the L4 microkernel [8]. NomadBIOS uses pre-copy migration to achieve very short best-case migration downtimes, but makes no attempt at adapting to the writable working set behavior of the migrating OS.

VMware has recently added OS migration support, dubbed *VMotion*, to their VirtualCenter management software. As this is commercial software and strictly disallows the publication of third-party benchmarks, we are only able to infer its behavior through VMware's own publications. These limitations make a thorough technical comparison impossible. However, based on the VirtualCenter User's Manual [9], we believe their approach is generally similar to ours and would expect it to perform to a similar standard.

Process migration, a hot topic in systems research during the 1980s [10, 11, 12, 13, 14], has seen very little use for real-world applications. Milojicic *et al* [2] give a thorough survey of possible reasons for this, including the problem of the *residual dependencies* that a migrated process retains on the machine from which it migrated. Examples of residual dependencies include open file descriptors, shared memory segments, and other local resources. These are undesirable because the original machine must remain available, and because they usually negatively impact the performance of migrated processes.

For example Sprite [15] processes executing on foreign nodes require some system calls to be forwarded to the home node for execution, leading to at best reduced performance and at worst widespread failure if the home node is unavailable. Although various efforts were made to ameliorate performance issues, the underlying reliance on the availability of the home node could not be avoided. A similar fragility occurs with MOSIX [14] where a deputy process on the home node must remain available to support remote execution.

We believe the residual dependency problem cannot easily be solved in any process migration scheme – even modern mobile run-times such as Java and .NET suffer from problems when network partition or machine crash causes class loaders to fail. The migration of entire operating systems inherently involves fewer or zero such dependencies, making it more resilient and robust.

3 Design

At a high level we can consider a virtual machine to encapsulate access to a set of physical resources. Providing live migration of these VMs in a clustered server environment leads us to focus on the physical resources used in such environments: specifically on memory, network and disk.

This section summarizes the design decisions that we have made in our approach to live VM migration. We start by describing how memory and then device access is moved across a set of physical hosts and then go on to a high-level description of how a migration progresses.

3.1 Migrating Memory

Moving the contents of a VM's memory from one physical host to another can be approached in any number of ways. However, when a VM is running a live service it is important that this transfer occurs in a manner that balances the requirements of minimizing both *downtime* and *total migration time*. The former is the period during which the service is unavailable due to there being no currently executing instance of the VM; this period will be directly visible to clients of the VM as service interruption. The latter is the duration between when migration is initiated and when the original VM may be finally discarded and, hence, the source host may potentially be taken down for maintenance, upgrade or repair.

It is easiest to consider the trade-offs between these requirements by generalizing memory transfer into three phases:

Push phase The source VM continues running while certain pages are pushed across the network to the new destination. To ensure consistency, pages modified during this process must be re-sent.

Stop-and-copy phase The source VM is stopped, pages are copied across to the destination VM, then the new VM is started.

Pull phase The new VM executes and, if it accesses a page that has not yet been copied, this page is faulted in ("pulled") across the network from the source VM.

Although one can imagine a scheme incorporating all three phases, most practical solutions select one or two of the three. For example, *pure stop-and-copy* [3, 4, 5] involves halting the original VM, copying all pages to the destination, and then starting the new VM. This has advantages in terms of simplicity but means that both downtime and total migration time are proportional to the amount of physical memory allocated to the VM. This can lead to an unacceptable outage if the VM is running a live service.

Another option is *pure demand-migration* [16] in which a short stop-and-copy phase transfers essential kernel data structures to the destination. The destination VM is then started, and other pages are transferred across the network on first use. This results in a much shorter downtime, but produces a much longer total migration time; and in practice, performance after migration is likely to be unacceptably degraded until a considerable set of pages have been faulted across. Until this time the VM will fault on a high proportion of its memory accesses, each of which initiates a synchronous transfer across the network.

The approach taken in this paper, *pre-copy* [11] migration, balances these concerns by combining a bounded iterative push phase with a typically very short stop-and-copy phase. By 'iterative' we mean that pre-copying occurs in *rounds*, in which the pages to be transferred during round n are those that are modified during round $n-1$ (all pages are transferred in the first round). Every VM will have some (hopefully small) set of pages that it updates very frequently and which are therefore poor candidates for pre-copy migration. Hence we bound the number of rounds of pre-copying, based on our analysis of the *writable working set* (WWS) behavior of typical server workloads, which we present in Section 4.

Finally, a crucial additional concern for live migration is the impact on active services. For instance, iteratively scanning and sending a VM's memory image between two hosts in a cluster could easily consume the entire bandwidth available between them and hence starve the active services of resources. This *service degradation* will occur to some extent during any live migration scheme. We address this issue by carefully controlling the network and CPU resources used by the migration process, thereby ensuring that it does not interfere excessively with active traffic or processing.

3.2 Local Resources

A key challenge in managing the migration of OS instances is what to do about resources that are associated with the physical machine that they are migrating away from. While memory can be copied directly to the new host, connections to local devices such as disks and network interfaces demand additional consideration. The two key problems that we have encountered in this space concern what to do with network resources and local storage.

For network resources, we want a migrated OS to maintain all open network connections without relying on forwarding mechanisms on the original host (which may be shut down following migration), or on support from mobility or redirection mechanisms that are not already present (as in [6]). A migrating VM will include all protocol state (e.g. TCP PCBs), and will carry its IP address with it.

To address these requirements we observed that in a cluster environment, the network interfaces of the source and destination machines typically exist on a single switched LAN. Our solution for managing migration with respect to network in this environment is to generate an unsolicited ARP reply from the migrated host, advertising that the IP has moved to a new location. This will reconfigure peers to send packets to the new physical address, and while a very small number of in-flight packets may be lost, the migrated domain will be able to continue using open connections with almost no observable interference.

Some routers are configured not to accept broadcast ARP replies (in order to prevent IP spoofing), so an unsolicited ARP may not work in all scenarios. If the operating system is aware of the migration, it can opt to send directed replies only to interfaces listed in its own ARP cache, to remove the need for a broadcast. Alternatively, on a switched network, the migrating OS can keep its original Ethernet MAC address, relying on the network switch to detect its move to a new port[1].

In the cluster, the migration of storage may be similarly addressed: Most modern data centers consolidate their storage requirements using a network-attached storage (NAS) device, in preference to using local disks in individual servers. NAS has many advantages in this environment, including simple centralised administration, widespread vendor support, and reliance on fewer spindles leading to a reduced failure rate. A further advantage for migration is that it obviates the need to migrate disk storage, as the NAS is uniformly accessible from all host machines in the cluster. We do not address the problem of migrating local-disk storage in this paper, although we suggest some possible strategies as part of our discussion of future work.

3.3 Design Overview

The logical steps that we execute when migrating an OS are summarized in Figure 1. We take a conservative approach to the management of migration with regard to safety and failure handling. Although the consequences of hardware failures can be severe, our basic principle is that safe migration should at no time leave a virtual OS more exposed

[1]Note that on most Ethernet controllers, hardware MAC filtering will have to be disabled if multiple addresses are in use (though some cards support filtering of multiple addresses in hardware) and so this technique is only practical for switched networks.

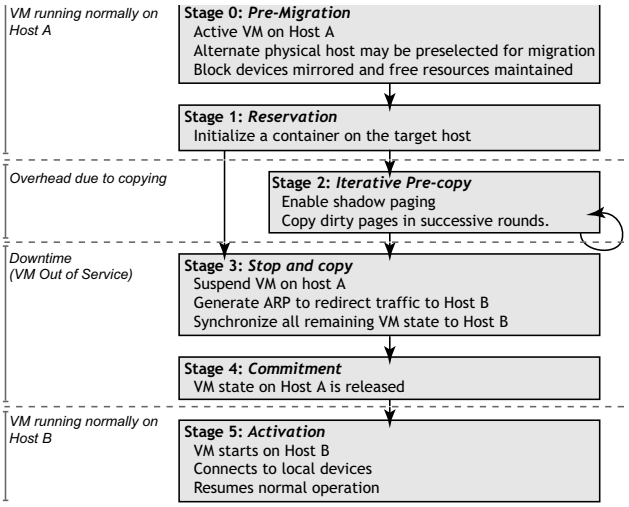


Figure 1: Migration timeline

to system failure than when it is running on the original single host. To achieve this, we view the migration process as a transactional interaction between the two hosts involved:

Stage 0: Pre-Migration We begin with an active VM on physical host A. To speed any future migration, a target host may be preselected where the resources required to receive migration will be guaranteed.

Stage 1: Reservation A request is issued to migrate an OS from host A to host B. We initially confirm that the necessary resources are available on B and reserve a VM container of that size. Failure to secure resources here means that the VM simply continues to run on A unaffected.

Stage 2: Iterative Pre-Copy During the first iteration, all pages are transferred from A to B. Subsequent iterations copy only those pages dirtied during the previous transfer phase.

Stage 3: Stop-and-Copy We suspend the running OS instance at A and redirect its network traffic to B. As described earlier, CPU state and any remaining inconsistent memory pages are then transferred. At the end of this stage there is a consistent suspended copy of the VM at both A and B. The copy at A is still considered to be primary and is resumed in case of failure.

Stage 4: Commitment Host B indicates to A that it has successfully received a consistent OS image. Host A acknowledges this message as commitment of the migration transaction: host A may now discard the original VM, and host B becomes the primary host.

Stage 5: Activation The migrated VM on B is now activated. Post-migration code runs to reattach device drivers to the new machine and advertise moved IP addresses.

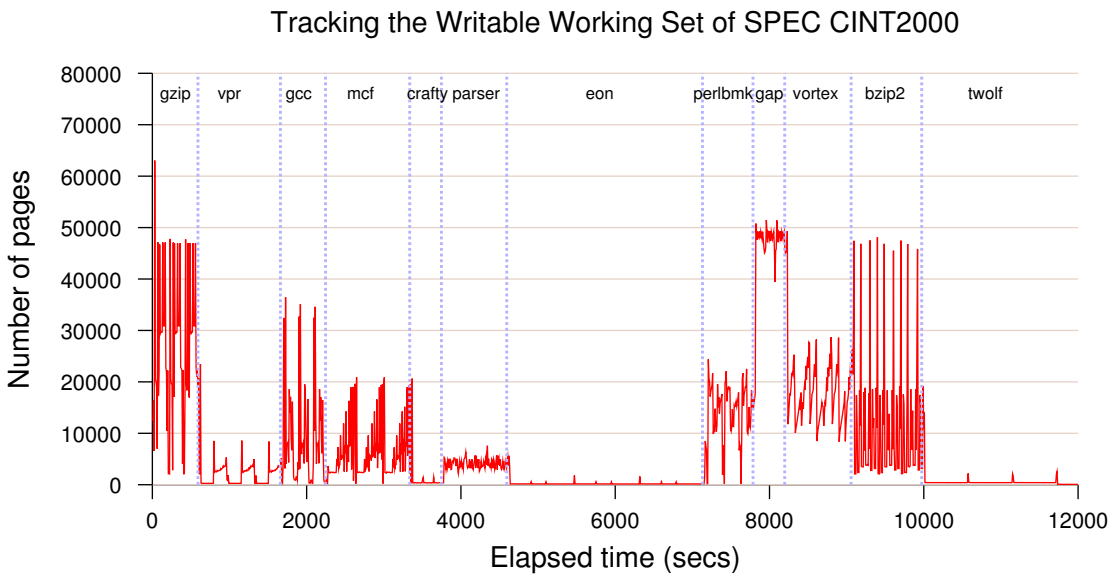


Figure 2: WWS curve for a complete run of SPEC CINT2000 (512MB VM)

This approach to failure management ensures that at least one host has a consistent VM image at all times during migration. It depends on the assumption that the original host remains stable until the migration commits, and that the VM may be suspended and resumed on that host with no risk of failure. Based on these assumptions, a migration request essentially attempts to move the VM to a new host, and on any sort of failure execution is resumed locally, aborting the migration.

4 Writable Working Sets

When migrating a live operating system, the most significant influence on service performance is the overhead of coherently transferring the virtual machine's memory image. As mentioned previously, a simple stop-and-copy approach will achieve this in time proportional to the amount of memory allocated to the VM. Unfortunately, during this time any running services are completely unavailable.

A more attractive alternative is pre-copy migration, in which the memory image is transferred while the operating system (and hence all hosted services) continue to run. The drawback however, is the wasted overhead of transferring memory pages that are subsequently modified, and hence must be transferred again. For many workloads there will be a small set of memory pages that are updated very frequently, and which it is not worth attempting to maintain coherently on the destination machine before stopping and copying the remainder of the VM.

The fundamental question for iterative pre-copy migration is: how does one determine when it is time to stop the pre-copy phase because too much time and resource is being wasted? Clearly if the VM being migrated never modifies memory, a single pre-copy of each memory page will suffice to transfer a consistent image to the destination. However, should the VM continuously dirty pages faster than the rate of copying, then all pre-copy work will be in vain and one should immediately stop and copy.

In practice, one would expect most workloads to lie somewhere between these extremes: a certain (possibly large) set of pages will seldom or never be modified and hence are good candidates for pre-copy, while the remainder will be written often and so should best be transferred via stop-and-copy – we dub this latter set of pages the *writable working set* (WWS) of the operating system by obvious extension of the original working set concept [17].

In this section we analyze the WWS of operating systems running a range of different workloads in an attempt to obtain some insight to allow us build heuristics for an efficient and controllable pre-copy implementation.

4.1 Measuring Writable Working Sets

To trace the writable working set behaviour of a number of representative workloads we used Xen's shadow page tables (see Section 5) to track dirtying statistics on all pages used by a particular executing operating system. This allows us to determine within any time period the set of pages written to by the virtual machine.

Using the above, we conducted a set of experiments to sam-

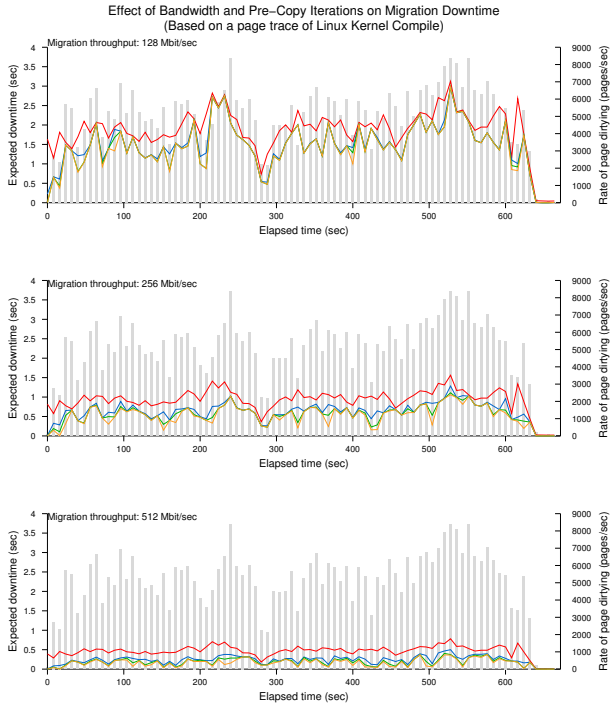


Figure 3: Expected downtime due to last-round memory copy on traced page dirtying of a Linux kernel compile.

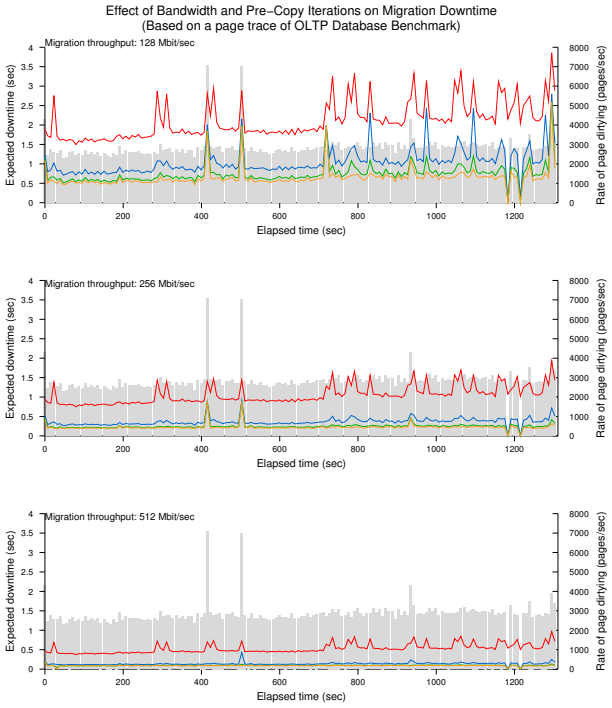


Figure 4: Expected downtime due to last-round memory copy on traced page dirtying of OLTP.

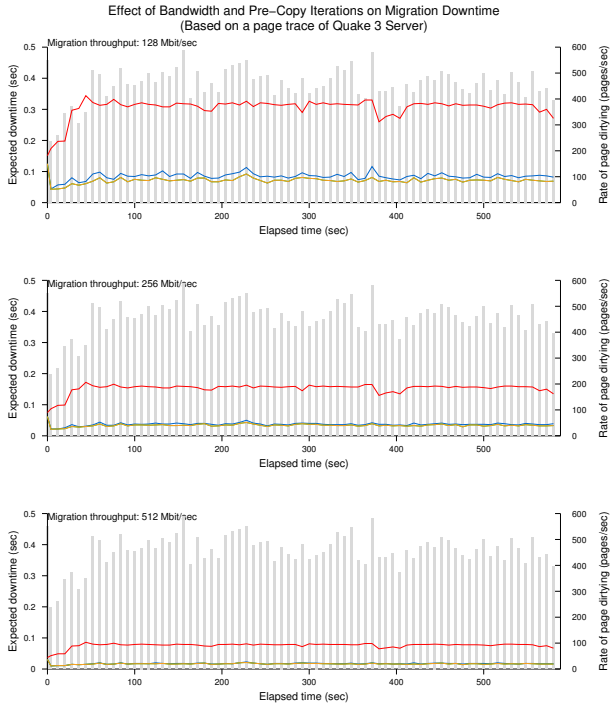


Figure 5: Expected downtime due to last-round memory copy on traced page dirtying of a Quake 3 server.

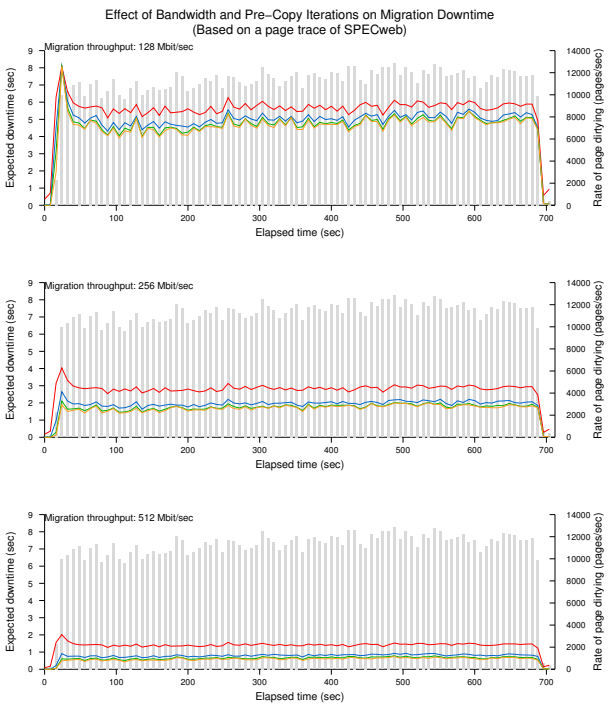


Figure 6: Expected downtime due to last-round memory copy on traced page dirtying of SPECweb.

ple the writable working set size for a variety of benchmarks. Xen was running on a dual processor Intel Xeon 2.4GHz machine, and the virtual machine being measured had a memory allocation of 512MB. In each case we started the relevant benchmark in one virtual machine and read the dirty bitmap every $50ms$ from another virtual machine, cleaning it every 8 seconds – in essence this allows us to compute the WWS with a (relatively long) 8 second window, but estimate it at a finer (50ms) granularity.

The benchmarks we ran were SPEC CINT2000, a Linux kernel compile, the OSDB OLTP benchmark using PostgreSQL and SPECweb99 using Apache. We also measured a Quake 3 server as we are particularly interested in highly interactive workloads.

Figure 2 illustrates the writable working set curve produced for the SPEC CINT2000 benchmark run. This benchmark involves running a series of smaller programs in order and measuring the overall execution time. The x-axis measures elapsed time, and the y-axis shows the number of 4KB pages of memory dirtied within the corresponding 8 second interval; the graph is annotated with the names of the sub-benchmark programs.

From this data we observe that the writable working set varies significantly between the different sub-benchmarks. For programs such as '`eon`' the WWS is a small fraction of the total working set and hence is an excellent candidate for migration. In contrast, '`gap`' has a consistently high dirtying rate and would be problematic to migrate. The other benchmarks go through various phases but are generally amenable to live migration. Thus performing a migration of an operating system will give different results depending on the workload and the precise moment at which migration begins.

4.2 Estimating Migration Effectiveness

We observed that we could use the trace data acquired to estimate the effectiveness of iterative pre-copy migration for various workloads. In particular we can simulate a particular network bandwidth for page transfer, determine how many pages would be dirtied during a particular iteration, and then repeat for successive iterations. Since we know the approximate WWS behaviour at every point in time, we can estimate the overall amount of data transferred in the final stop-and-copy round and hence estimate the downtime.

Figures 3–6 show our results for the four remaining workloads. Each figure comprises three graphs, each of which corresponds to a particular network bandwidth limit for page transfer; each individual graph shows the WWS histogram (in light gray) overlaid with four line plots estimating service downtime for up to four pre-copying rounds.

Looking at the topmost line (one pre-copy iteration), the first thing to observe is that pre-copy migration always performs considerably better than naive stop-and-copy. For a 512MB virtual machine this latter approach would require 32, 16, and 8 seconds downtime for the 128Mbit/sec, 256Mbit/sec and 512Mbit/sec bandwidths respectively. Even in the worst case (the starting phase of SPECweb), a single pre-copy iteration reduces downtime by a factor of four. In most cases we can expect to do considerably better – for example both the Linux kernel compile and the OLTP benchmark typically experience a reduction in downtime of at least a factor of sixteen.

The remaining three lines show, in order, the effect of performing a total of two, three or four pre-copy iterations prior to the final stop-and-copy round. In most cases we see an increased reduction in downtime from performing these additional iterations, although with somewhat diminishing returns, particularly in the higher bandwidth cases.

This is because all the observed workloads exhibit a small but extremely frequently updated set of 'hot' pages. In practice these pages will include the stack and local variables being accessed within the currently executing processes as well as pages being used for network and disk traffic. The hottest pages will be dirtied at least as fast as we can transfer them, and hence must be transferred in the final stop-and-copy phase. This puts a lower bound on the best possible service downtime for a particular benchmark, network bandwidth and migration start time.

This interesting tradeoff suggests that it may be worthwhile increasing the amount of bandwidth used for page transfer in later (and shorter) pre-copy iterations. We will describe our rate-adaptive algorithm based on this observation in Section 5, and demonstrate its effectiveness in Section 6.

5 Implementation Issues

We designed and implemented our pre-copying migration engine to integrate with the Xen virtual machine monitor [1]. Xen securely divides the resources of the host machine amongst a set of resource-isolated virtual machines each running a dedicated OS instance. In addition, there is one special *management virtual machine* used for the administration and control of the machine.

We considered two different methods for initiating and managing state transfer. These illustrate two extreme points in the design space: *managed migration* is performed largely outside the migratee, by a migration daemon running in the management VM; in contrast, *self migration* is implemented almost entirely within the migratee OS with only a small stub required on the destination machine.

In the following sections we describe some of the implementation details of these two approaches. We describe how we use dynamic network rate-limiting to effectively

balance network contention against OS downtime. We then proceed to describe how we ameliorate the effects of rapid page dirtying, and describe some performance enhancements that become possible when the OS is aware of its migration — either through the use of self migration, or by adding explicit paravirtualization interfaces to the VMM.

5.1 Managed Migration

Managed migration is performed by migration daemons running in the management VMs of the source and destination hosts. These are responsible for creating a new VM on the destination machine, and coordinating transfer of live system state over the network.

When transferring the memory image of the still-running OS, the control software performs *rounds* of copying in which it performs a complete scan of the VM's memory pages. Although in the first round all pages are transferred to the destination machine, in subsequent rounds this copying is restricted to pages that were dirtied during the previous round, as indicated by a *dirty bitmap* that is copied from Xen at the start of each round.

During normal operation the page tables managed by each guest OS are the ones that are walked by the processor's MMU to fill the TLB. This is possible because guest OSes are exposed to real physical addresses and so the page tables they create do not need to be mapped to physical addresses by Xen.

To log pages that are dirtied, Xen inserts *shadow page tables* underneath the running OS. The shadow tables are populated on demand by translating sections of the guest page tables. Translation is very simple for dirty logging: all page-table entries (PTEs) are initially read-only mappings in the shadow tables, regardless of what is permitted by the guest tables. If the guest tries to modify a page of memory, the resulting page fault is trapped by Xen. If write access is permitted by the relevant guest PTE then this permission is extended to the shadow PTE. At the same time, we set the appropriate bit in the VM's dirty bitmap.

When the bitmap is copied to the control software at the start of each pre-copying round, Xen's bitmap is cleared and the shadow page tables are destroyed and recreated as the migratee OS continues to run. This causes all write permissions to be lost: all pages that are subsequently updated are then added to the now-clear dirty bitmap.

When it is determined that the pre-copy phase is no longer beneficial, using heuristics derived from the analysis in Section 4, the OS is sent a control message requesting that it suspend itself in a state suitable for migration. This causes the OS to prepare for resumption on the destination machine; Xen informs the control software once the OS has done this. The dirty bitmap is scanned one last time for remaining inconsistent memory pages, and these are transferred to the destination together with the VM's checkpointed CPU-register state.

Once this final information is received at the destination, the VM state on the source machine can safely be discarded. Control software on the destination machine scans the memory map and rewrites the guest's page tables to reflect the addresses of the memory pages that it has been allocated. Execution is then resumed by starting the new VM at the point that the old VM checkpointed itself. The OS then restarts its virtual device drivers and updates its notion of wallclock time.

Since the transfer of pages is OS agnostic, we can easily support any guest operating system – all that is required is a small paravirtualized stub to handle resumption. Our implementation currently supports Linux 2.4, Linux 2.6 and NetBSD 2.0.

5.2 Self Migration

In contrast to the managed method described above, self migration [18] places the majority of the implementation within the OS being migrated. In this design no modifications are required either to Xen or to the management software running on the source machine, although a migration stub must run on the destination machine to listen for incoming migration requests, create an appropriate empty VM, and receive the migrated system state.

The pre-copying scheme that we implemented for self migration is conceptually very similar to that for managed migration. At the start of each pre-copying round every page mapping in every virtual address space is write-protected. The OS maintains a dirty bitmap tracking dirtied physical pages, setting the appropriate bits as write faults occur. To discriminate migration faults from other possible causes (for example, copy-on-write faults, or access-permission faults) we reserve a spare bit in each PTE to indicate that it is write-protected only for dirty-logging purposes.

The major implementation difficulty of this scheme is to transfer a consistent OS checkpoint. In contrast with a managed migration, where we simply suspend the migratee to obtain a consistent checkpoint, self migration is far harder because the OS must continue to run in order to transfer its final state. We solve this difficulty by logically checkpointing the OS on entry to a final *two-stage* stop-and-copy phase. The first stage disables all OS activity except for migration and then peforms a final scan of the dirty bitmap, clearing the appropriate bit as each page is transferred. Any pages that are dirtied during the final scan, and that are still marked as dirty in the bitmap, are copied to a shadow buffer. The second and final stage then transfers the contents of the shadow buffer — page updates are ignored during this transfer.

5.3 Dynamic Rate-Limiting

It is not always appropriate to select a single network bandwidth limit for migration traffic. Although a low limit avoids impacting the performance of running services, analysis in Section 4 showed that we must eventually pay in the form of an extended downtime because the hottest pages in the writable working set are not amenable to pre-copy migration. The downtime can be reduced by increasing the bandwidth limit, albeit at the cost of additional network contention.

Our solution to this impasse is to dynamically adapt the bandwidth limit during each pre-copying round. The administrator selects a minimum and a maximum bandwidth limit. The first pre-copy round transfers pages at the minimum bandwidth. Each subsequent round counts the number of pages dirtied in the previous round, and divides this by the duration of the previous round to calculate the *dirtying rate*. The bandwidth limit for the next round is then determined by adding a constant increment to the previous round's dirtying rate — we have empirically determined that 50Mbit/sec is a suitable value. We terminate pre-copying when the calculated rate is greater than the administrator's chosen maximum, or when less than 256KB remains to be transferred. During the final stop-and-copy phase we minimize service downtime by transferring memory at the maximum allowable rate.

As we will show in Section 6, using this adaptive scheme results in the bandwidth usage remaining low during the transfer of the majority of the pages, increasing only at the end of the migration to transfer the hottest pages in the WWS. This effectively balances short downtime with low average network contention and CPU usage.

5.4 Rapid Page Dirtying

Our working-set analysis in Section 4 shows that every OS workload has some set of pages that are updated extremely frequently, and which are therefore not good candidates for pre-copy migration even when using all available network bandwidth. We observed that rapidly-modified pages are very likely to be dirtied again by the time we attempt to transfer them in any particular pre-copying round. We therefore periodically 'peek' at the current round's dirty bitmap and transfer only those pages dirtied in the previous round that have not been dirtied again at the time we scan them.

We further observed that page dirtying is often physically *clustered* — if a page is dirtied then it is disproportionally likely that a close neighbour will be dirtied soon after. This increases the likelihood that, if our peeking does not detect one page in a cluster, it will detect none. To avoid this unfortunate behaviour we scan the VM's physical memory space in a pseudo-random order.

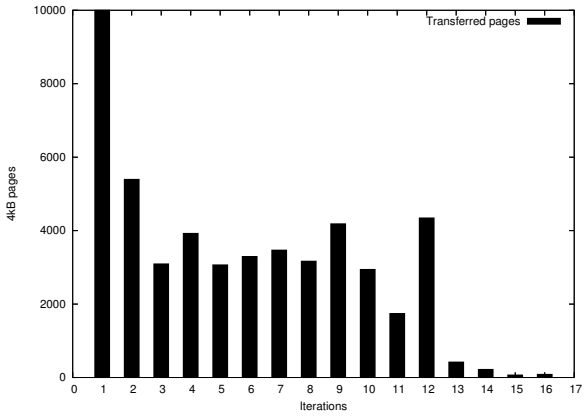


Figure 7: Rogue-process detection during migration of a Linux kernel build. After the twelfth iteration a maximum limit of forty write faults is imposed on every process, drastically reducing the total writable working set.

5.5 Paravirtualized Optimizations

One key benefit of paravirtualization is that operating systems can be made aware of certain important differences between the real and virtual environments. In terms of migration, this allows a number of optimizations by informing the operating system that it is about to be migrated – at this stage a migration stub handler within the OS could help improve performance in at least the following ways:

Stunning Rogue Processes. Pre-copy migration works best when memory pages can be copied to the destination host faster than they are dirtied by the migrating virtual machine. This may not always be the case – for example, a test program which writes one word in every page was able to dirty memory at a rate of 320 Gbit/sec, well ahead of the transfer rate of any Ethernet interface. This is a synthetic example, but there may well be cases in practice in which pre-copy migration is unable to keep up, or where migration is prolonged unnecessarily by one or more 'rogue' applications.

In both the managed and self migration cases, we can mitigate against this risk by forking a monitoring thread within the OS kernel when migration begins. As it runs within the OS, this thread can monitor the WWS of *individual processes* and take action if required. We have implemented a simple version of this which simply limits each process to 40 write faults before being moved to a wait queue – in essence we 'stun' processes that make migration difficult. This technique works well, as shown in Figure 7, although

one must be careful not to stun important interactive services.

Freeing Page Cache Pages. A typical operating system will have a number of 'free' pages at any time, ranging from truly free (page allocator) to cold buffer cache pages. When informed a migration is to begin, the OS can simply return some or all of these pages to Xen in the same way it would when using the ballooning mechanism described in [1]. This means that the time taken for the first "full pass" iteration of pre-copy migration can be reduced, sometimes drastically. However should the contents of these pages be needed again, they will need to be faulted back in from disk, incurring greater overall cost.

6 Evaluation

In this section we present a thorough evaluation of our implementation on a wide variety of workloads. We begin by describing our test setup, and then go on explore the migration of several workloads in detail. Note that none of the experiments in this section use the paravirtualized optimizations discussed above since we wished to measure the baseline performance of our system.

6.1 Test Setup

We perform test migrations between an identical pair of Dell PE-2650 server-class machines, each with dual Xeon 2GHz CPUs and 2GB memory. The machines have Broadcom TG3 network interfaces and are connected via switched Gigabit Ethernet. In these experiments only a single CPU was used, with HyperThreading enabled. Storage is accessed via the iSCSI protocol from an NetApp F840 network attached storage server except where noted otherwise. We used XenLinux 2.4.27 as the operating system in all cases.

6.2 Simple Web Server

We begin our evaluation by examining the migration of an Apache 1.3 web server serving static content at a high rate. Figure 8 illustrates the throughput achieved when continuously serving a single 512KB file to a set of one hundred concurrent clients. The web server virtual machine has a memory allocation of 800MB.

At the start of the trace, the server achieves a consistent throughput of approximately 870Mbit/sec. Migration starts twenty seven seconds into the trace but is initially rate-limited to 100Mbit/sec (12% CPU), resulting in the server throughput dropping to 765Mbit/s. This initial low-rate pass transfers 776MB and lasts for 62 seconds, at which point the migration algorithm described in Section 5 increases its rate over several iterations and finally suspends the VM after a further 9.8 seconds. The final stop-and-copy phase then transfer the remaining pages and the web server resumes at full rate after a 165ms outage.

This simple example demonstrates that a highly loaded server can be migrated with both controlled impact on live services and a short downtime. However, the working set of the server in this case is rather small, and so this should be expected to be a relatively easy case for live migration.

6.3 Complex Web Workload: SPECweb99

A more challenging Apache workload is presented by SPECweb99, a complex application-level benchmark for evaluating web servers and the systems that host them. The workload is a complex mix of page requests: 30% require dynamic content generation, 16% are HTTP POST operations, and 0.5% execute a CGI script. As the server runs, it generates access and POST logs, contributing to disk (and therefore network) throughput.

A number of client machines are used to generate the load for the server under test, with each machine simulating a collection of users concurrently accessing the web site. SPECweb99 defines a minimum quality of service that each user must receive for it to count as 'conformant'; an aggregate bandwidth in excess of 320Kbit/sec over a series of requests. The SPECweb score received is the number of conformant users that the server successfully maintains. The considerably more demanding workload of SPECweb represents a challenging candidate for migration.

We benchmarked a single VM running SPECweb and recorded a maximum score of 385 conformant clients — we used the RedHat `gnbd` network block device in place of iSCSI as the lighter-weight protocol achieves higher performance. Since at this point the server is effectively in overload, we then relaxed the offered load to 90% of maximum (350 conformant connections) to represent a more realistic scenario.

Using a virtual machine configured with 800MB of memory, we migrated a SPECweb99 run in the middle of its execution. Figure 9 shows a detailed analysis of this migration. The x-axis shows time elapsed since start of migration, while the y-axis shows the network bandwidth being used to transfer pages to the destination. Darker boxes illustrate the page transfer process while lighter boxes show the pages dirtied during each iteration. Our algorithm adjusts the transfer rate relative to the page dirty rate observed during the previous round (denoted by the height of the lighter boxes).

As in the case of the static web server, migration begins

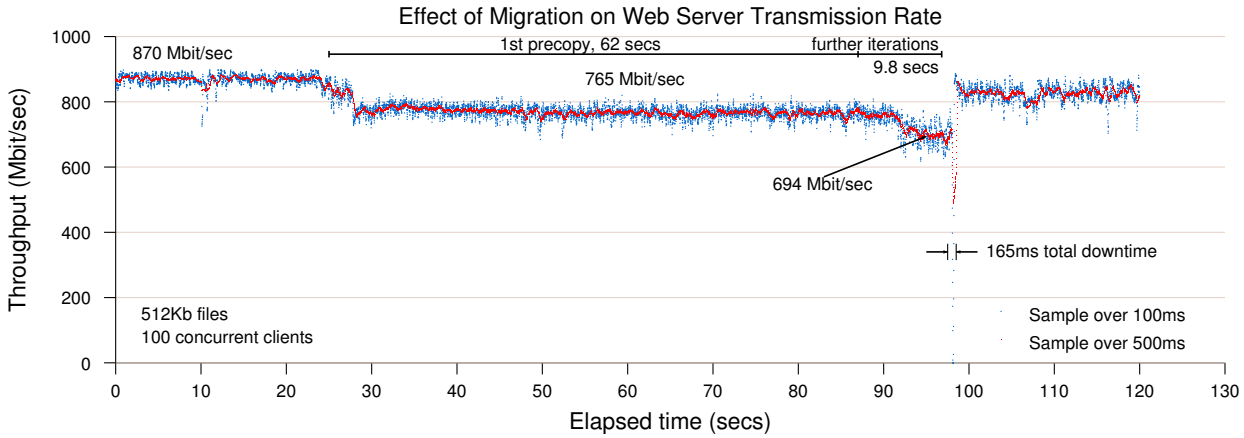


Figure 8: Results of migrating a running web server VM.

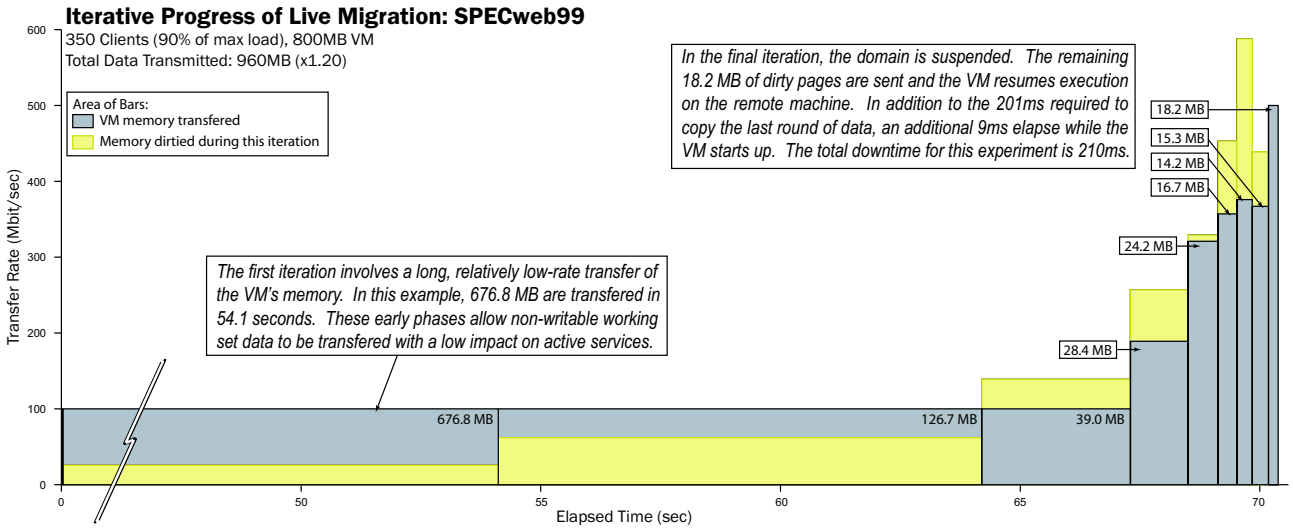


Figure 9: Results of migrating a running SPECweb VM.

with a long period of low-rate transmission as a first pass is made through the memory of the virtual machine. This first round takes 54.1 seconds and transmits 676.8MB of memory. Two more low-rate rounds follow, transmitting 126.7MB and 39.0MB respectively before the transmission rate is increased.

The remainder of the graph illustrates how the adaptive algorithm tracks the page dirty rate over successively shorter iterations before finally suspending the VM. When suspension takes place, 18.2MB of memory remains to be sent. This transmission takes $201ms$, after which an additional $9ms$ is required for the domain to resume normal execution.

The total downtime of $210ms$ experienced by the SPECweb clients is sufficiently brief to maintain the 350 conformant clients. This result is an excellent validation of our approach: a heavily (90% of maximum) loaded server is migrated to a separate physical host with a total migration time of seventy-one seconds. Furthermore the migration does not interfere with the quality of service demanded by SPECweb's workload. This illustrates the applicability of migration as a tool for administrators of demanding live services.

6.4 Low-Latency Server: Quake 3

Another representative application for hosting environments is a multiplayer on-line game server. To determine the effectiveness of our approach in this case we configured a virtual machine with 64MB of memory running a

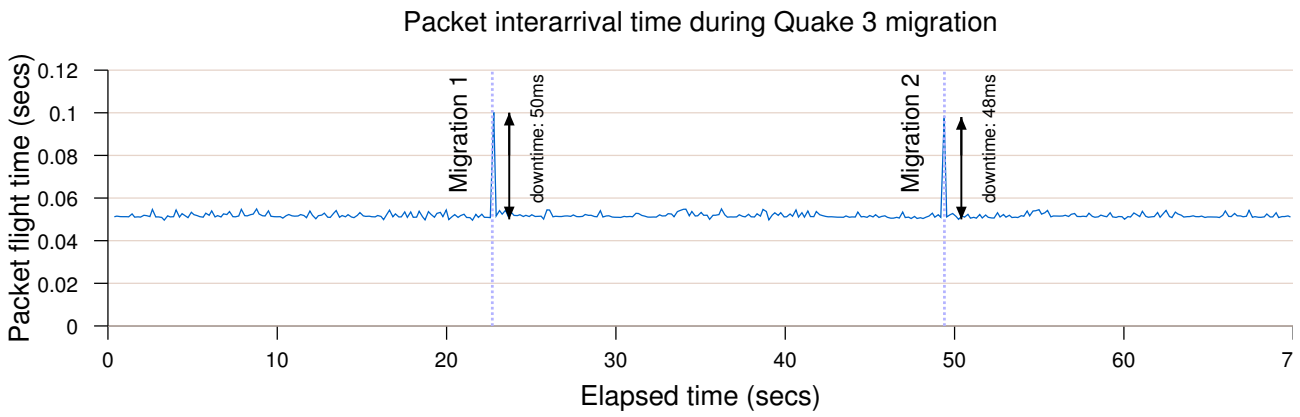


Figure 10: Effect on packet response time of migrating a running Quake 3 server VM.

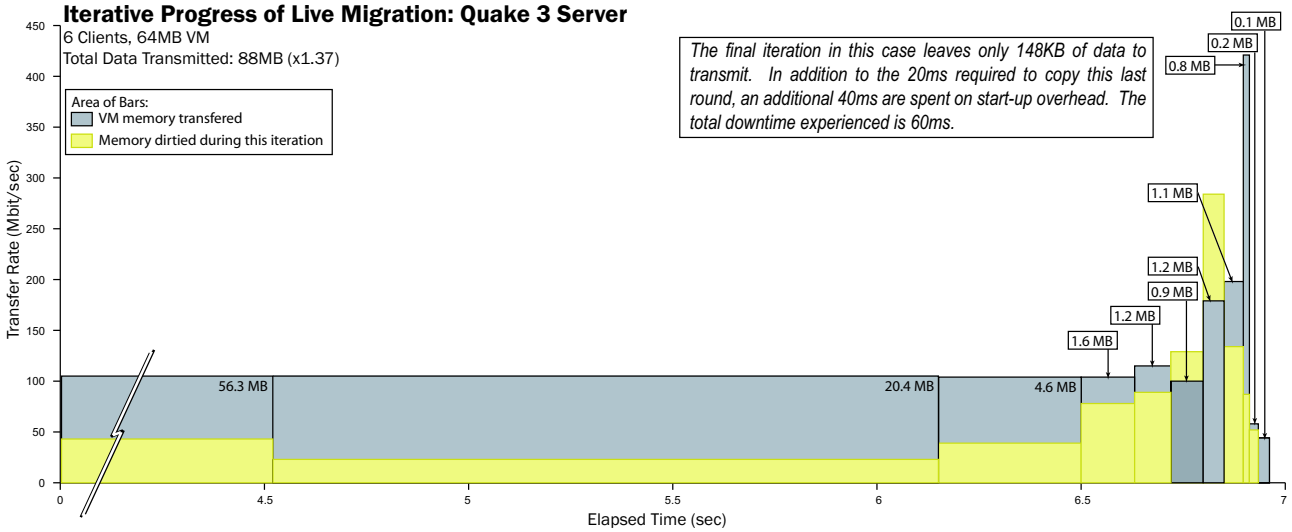


Figure 11: Results of migrating a running Quake 3 server VM.

Quake 3 server. Six players joined the game and started to play within a shared arena, at which point we initiated a migration to another machine. A detailed analysis of this migration is shown in Figure 11.

The trace illustrates a generally similar progression as for SPECweb, although in this case the amount of data to be transferred is significantly smaller. Once again the transfer rate increases as the trace progresses, although the final stop-and-copy phase transfers so little data (148KB) that the full bandwidth is not utilized.

Overall, we are able to perform the live migration with a total downtime of $60ms$. To determine the effect of migration on the live players, we performed an additional experiment in which we migrated the running Quake 3 server twice and measured the inter-arrival time of packets received by clients. The results are shown in Figure 10. As can be seen, from the client point of view migration manifests itself as a transient increase in response time of $50ms$. In neither case was this perceptible to the players.

6.5 A Diabolical Workload: MMuncher

As a final point in our evaluation, we consider the situation in which a virtual machine is writing to memory faster than can be transferred across the network. We test this diabolical case by running a 512MB host with a simple C program that writes constantly to a 256MB region of memory. The results of this migration are shown in Figure 12.

In the first iteration of this workload, we see that half of the memory has been transmitted, while the other half is immediately marked dirty by our test program. Our algorithm attempts to adapt to this by scaling itself relative to the perceived initial rate of dirtying; this scaling proves in-

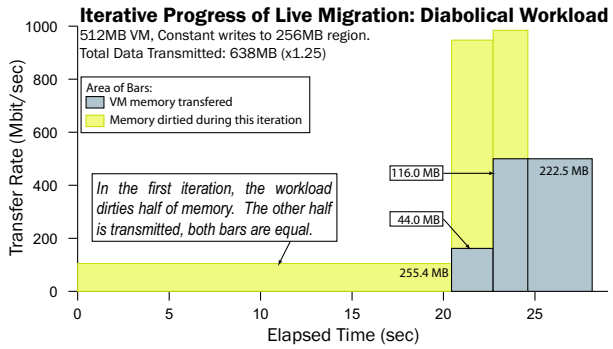


Figure 12: Results of migrating a VM running a diabolical workload.

sufficient, as the rate at which the memory is being written becomes apparent. In the third round, the transfer rate is scaled up to 500Mbit/s in a final attempt to outpace the memory writer. As this last attempt is still unsuccessful, the virtual machine is suspended, and the remaining dirty pages are copied, resulting in a downtime of 3.5 seconds. Fortunately such dirtying rates appear to be rare in real workloads.

7 Future Work

Although our solution is well-suited for the environment we have targeted – a well-connected data-center or cluster with network-accessed storage – there are a number of areas in which we hope to carry out future work. This would allow us to extend live migration to wide-area networks, and to environments that cannot rely solely on network-attached storage.

7.1 Cluster Management

In a cluster environment where a pool of virtual machines are hosted on a smaller set of physical servers, there are great opportunities for dynamic load balancing of processor, memory and networking resources. A key challenge is to develop cluster control software which can make informed decision as to the placement and movement of virtual machines.

A special case of this is 'evacuating' VMs from a node that is to be taken down for scheduled maintenance. A sensible approach to achieving this is to migrate the VMs in increasing order of their observed WWS. Since each VM migrated frees resources on the node, additional CPU and network becomes available for those VMs which need it most. We are in the process of building a cluster controller for Xen systems.

7.2 Wide Area Network Redirection

Our layer 2 redirection scheme works efficiently and with remarkably low outage on modern gigabit networks. However, when migrating outside the local subnet this mechanism will not suffice. Instead, either the OS will have to obtain a new IP address which is within the destination subnet, or some kind of indirection layer, on top of IP, must exist. Since this problem is already familiar to laptop users, a number of different solutions have been suggested. One of the more prominent approaches is that of Mobile IP [19] where a node on the home network (the *home agent*) forwards packets destined for the client (*mobile node*) to a *care-of address* on the foreign network. As with all residual dependencies this can lead to both performance problems and additional failure modes.

Snoeren and Balakrishnan [20] suggest addressing the problem of connection migration at the TCP level, augmenting TCP with a secure token negotiated at connection time, to which a relocated host can refer in a special SYN packet requesting reconnection from a new IP address. Dynamic DNS updates are suggested as a means of locating hosts after a move.

7.3 Migrating Block Devices

Although NAS prevails in the modern data center, some environments may still make extensive use of local disks. These present a significant problem for migration as they are usually considerably larger than volatile memory. If the entire contents of a disk must be transferred to a new host before migration can complete, then total migration times may be intolerably extended.

This latency can be avoided at migration time by arranging to *mirror* the disk contents at one or more remote hosts. For example, we are investigating using the built-in software RAID and iSCSI functionality of Linux to implement disk mirroring before and during OS migration. We imagine a similar use of software RAID-5, in cases where data on disks requires a higher level of availability. Multiple hosts can act as storage targets for one another, increasing availability at the cost of some network traffic.

The effective management of local storage for clusters of virtual machines is an interesting problem that we hope to further explore in future work. As virtual machines will typically work from a small set of common system images (for instance a generic Fedora Linux installation) and make individual changes above this, there seems to be opportunity to manage copy-on-write system images across a cluster in a way that facilitates migration, allows replication, and makes efficient use of local disks.

8 Conclusion

By integrating live OS migration into the Xen virtual machine monitor we enable rapid movement of interactive workloads within clusters and data centers. Our dynamic network-bandwidth adaptation allows migration to proceed with minimal impact on running services, while reducing total downtime to below discernable thresholds.

Our comprehensive evaluation shows that realistic server workloads such as SPECweb99 can be migrated with just $210ms$ downtime, while a Quake3 game server is migrated with an imperceptible $60ms$ outage.

References

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating Systems Principles (SOSP19)*, pages 164–177. ACM Press, 2003.

[2] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. *ACM Computing Surveys*, 32(3):241–299, 2000.

[3] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M.Rosenblum. Optimizing the migration of virtual computers. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation (OSDI-02)*, December 2002.

[4] M. Kozuch and M. Satyanarayanan. Internet suspend/resume. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, 2002.

[5] Andrew Whitaker, Richard S. Cox, Marianne Shaw, and Steven D. Gribble. Constructing services with interposable virtual hardware. In *Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI '04)*, 2004.

[6] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of zap: A system for migrating computing environments. In *Proc. 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI-02)*, pages 361–376, December 2002.

[7] Jacob G. Hansen and Asger K. Henriksen. Nomadic operating systems. Master's thesis, Dept. of Computer Science, University of Copenhagen, Denmark, 2002.

[8] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, and Sebastian Schönberg. The performance of micro-kernel-based systems. In *Proceedings of the sixteenth ACM Symposium on Operating System Principles*, pages 66–77. ACM Press, 1997.

[9] VMWare, Inc. *VMWare VirtualCenter Version 1.2 User's Manual*. 2004.

[10] Michael L. Powell and Barton P. Miller. Process migration in DEMOS/MP. In *Proceedings of the ninth ACM Symposium on Operating System Principles*, pages 110–119. ACM Press, 1983.

[11] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the V-system. In *Proceedings of the tenth ACM Symposium on Operating System Principles*, pages 2–12. ACM Press, 1985.

[12] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. *ACM Trans. Comput. Syst.*, 6(1):109–133, 1988.

[13] Fred Douglis and John K. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software - Practice and Experience*, 21(8):757–785, 1991.

[14] A. Barak and O. La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, 13(4-5):361–372, March 1998.

[15] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite network operating system. *Computer Magazine of the Computer Group News of the IEEE Computer Group Society, ; ACM CR 8905-0314*, 21(2), 1988.

[16] E. Zayas. Attacking the process migration bottleneck. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 13–24. ACM Press, 1987.

[17] Peter J. Denning. Working Sets Past and Present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, January 1980.

[18] Jacob G. Hansen and Eric Jul. Self-migration of operating systems. In *Proceedings of the 11th ACM SIGOPS European Workshop (EW 2004)*, pages 126–130, 2004.

[19] C. E. Perkins and A. Myles. Mobile IP. *Proceedings of International Telecommunications Symposium*, pages 415–419, 1997.

[20] Alex C. Snoeren and Hari Balakrishnan. An end-to-end approach to host mobility. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 155–166. ACM Press, 2000.

Botz-4-Sale: Surviving Organized DDoS Attacks That Mimic Flash Crowds

Srikanth Kandula Dina Katabi
MIT
{kandula,dina}@csail.mit.edu

Matthias Jacob
Princeton
mjacob@princeton.edu

Arthur Berger
MIT/Akamai
awberger@mit.edu

Abstract– Recent denial of service attacks are mounted by professionals using Botnets of tens of thousands of compromised machines. To circumvent detection, attackers are increasingly moving away from bandwidth floods to attacks that mimic the Web browsing behavior of a large number of clients, and target expensive higher-layer resources such as CPU, database and disk bandwidth. The resulting attacks are hard to defend against using standard techniques, as the malicious requests differ from the legitimate ones in intent but not in content.

We present the design and implementation of Kill-Bots, a kernel extension to protect Web servers against DDoS attacks that masquerade as flash crowds. Kill-Bots provides authentication using graphical tests but is different from other systems that use graphical tests. First, Kill-Bots uses an intermediate stage to identify the IP addresses that ignore the test, and persistently bombard the server with requests despite repeated failures at solving the tests. These machines are bots because their intent is to congest the server. Once these machines are identified, Kill-Bots blocks their requests, turns the graphical tests off, and allows access to legitimate users who are unable or unwilling to solve graphical tests. Second, Kill-Bots sends a test and checks the client's answer without allowing unauthenticated clients access to sockets, TCBs, and worker processes. Thus, it protects the authentication mechanism from being DDoSed. Third, Kill-Bots combines authentication with admission control. As a result, it improves performance, regardless of whether the server overload is caused by DDoS or a true Flash Crowd.

1 Introduction

Denial of service attacks are increasingly mounted by professionals in exchange for money or material benefits [35]. Botnets of thousands of compromised machines are rented by the hour on IRC and used to DDoS online businesses to extort money or obtain commercial advantages [17, 26, 45]. The DDoS business is thriving; increasingly aggressive worms can infect up to 30,000 new machines per day. These zombies/bots are then used for DDoS and other attacks [17, 43]. Recently, a Massachusetts businessman paid members of the computer underground to launch organized, crippling DDoS attacks against three of his competitors [35]. The attackers used Botnets of more than 10,000 machines. When the simple SYN flood failed, they launched an HTTP flood, pulling large image files from the victim server in overwhelming numbers. At its peak, the onslaught allegedly kept the victim company offline for two weeks. In another instance, attackers ran a massive number of queries through the victim's search engine, bringing the server down [35].

To circumvent detection, attackers are increasingly moving away from pure bandwidth floods to stealthy DDoS attacks that masquerade as flash crowds. They profile the victim server and mimic legitimate Web browsing behavior of a large number of clients. These attacks target higher layer server resources like sockets, disk bandwidth, database bandwidth and worker processes [13, 24, 35]. We call such DDoS attacks CyberSlam, after the first FBI case involving DDoS-for-hire [35]. The MyDoom worm [13], many DDoS extortion attacks [24], and recent DDoS-for-hire attacks are all instances of CyberSlam [12, 24, 35].

Countering CyberSlam is a challenge because the requests originating from the zombies are indistinguishable from the requests generated by legitimate users. The malicious requests differ from the legitimate ones in intent but not in content. The malicious requests arrive from a large number of geographically distributed machines; thus they cannot be filtered on the IP prefix. Also, many sites do not use passwords or login information, and even when they do, passwords could be easily stolen from the hard disk of a compromised machine. Further, checking the site-specific password requires establishing a connection and allowing unauthenticated clients to access socket buffers, TCBs, and worker processes, making it easy to mount an attack on the authentication mechanism itself. Defending against CyberSlam using computational puzzles, which require the client to perform heavy computation before accessing the site, is not effective because computing power is usually abundant in a Botnet. Finally, in contrast to bandwidth attacks [27, 40], it is difficult to detect big resource consumers when the attack targets higher-layer bottlenecks such as CPU, database, and disk because commodity operating systems do not support fine-grained resource monitoring [15, 48]. Further, an attacker can resort to mutating attacks which cycle between different bottlenecks [25].

This paper proposes Kill-Bots, a kernel extension that protects Web servers against CyberSlam attacks. It is targeted towards small or medium online businesses as well as non-commercial Web sites. Kill-Bots combines two

functions: authentication and admission control.

(a) Authentication: The authentication mechanism is activated when the server is overloaded. It has 2 stages:

- In $Stage_1$, Kill-Bots requires each new session to solve a reverse Turing test to obtain access to the server. Humans can easily solve a reverse Turing test, but zombies cannot. We focus on graphical tests [47], though Kill-Bots works with other types of reverse Turing tests. Legitimate clients either solve the test, or try to reload a few times and, if they still cannot access the server, decide to come back later. In contrast, the zombies which want to congest the server continue sending new requests without solving the test. Kill-Bots uses this difference in behavior between legitimate users and zombies to identify the IP addresses that belong to zombies and drop their requests. Kill-Bots uses SYN cookies to prevent spoofing of IP addresses and a Bloom filter to count how often an IP address failed to solve a test. It discards requests from a client if the number of its unsolved tests exceeds a given threshold (e.g., 32).
- Kill-Bots switches to $Stage_2$ after the set of detected zombie IP addresses stabilizes (i.e., the filter does not learn any new bad IP addresses). In this stage, tests are no longer served. Instead, Kill-Bots relies solely on the Bloom filter to drop requests from malicious clients. This allows legitimate users who cannot, or do not want to solve graphical puzzles access to the server despite the ongoing attack.

(b) Admission Control: Kill-Bots combines authentication with admission control. A Web site that performs authentication to protect itself from DDoS encounters a general problem: It has a certain pool of resources, which it needs to divide between authenticating new arrivals and servicing clients that are already authenticated. Devoting excess resources to authentication might leave the server unable to fully serve the authenticated clients, and hence, wastes server resources on authenticating new clients that it cannot serve. On the other hand, devoting excess resources to serving authenticated clients reduces the rate at which new clients are authenticated and admitted, leading to idle periods with no clients in service.

Kill-Bots computes the admission probability α that maximizes the server's goodput (i.e., the optimal probability with which new clients should be authenticated). It also provides a controller that allows the server to converge to the desired admission probability using simple measurements of the server's utilization. Admission control is a standard mechanism for combating server overload [18, 46, 48], but Kill-Bots examines admission control in the context of malicious clients and connects it with client authentication.

Fig. 1 summarizes Kill-Bots. When a new connection arrives, it is first checked against the list of detected zombie addresses. If the IP address is not recognized as a zombie, Kill-Bots admits the connection with probability $\alpha = f(load)$. In $Stage_1$, admitted connections are served a graphical puzzle. If the client solves the puzzle, it is given a Kill-Bots HTTP cookie which allows its future connections, for a short period, to access the server without being subject to admission control and without having to solve new puzzles. In $Stage_2$, Kill-Bots no longer issues puzzles; admitted connections are immediately given a Kill-Bots HTTP cookie.

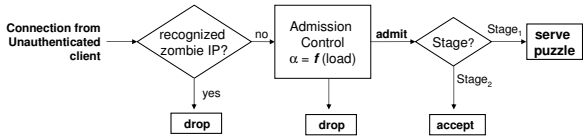


Figure 1: Kill-Bots Overview. Note that graphical puzzles are only served during $Stage_1$.

Kill-Bots has a few important characteristics.

- **Kill-Bots addresses graphical tests' bias against users who are unable or unwilling to solve them.** Prior work that employs graphical tests ignores the resulting user inconvenience as well as their bias against blind and inexperienced humans [32]. Kill-Bots is the first system to employ graphical tests to distinguish humans from automated zombies, while limiting their negative impact on legitimate users who cannot or do not want to solve them.
- **Kill-Bots sends a puzzle without giving access to TCBs or socket buffers.** Typically, sending the client a puzzle requires establishing a connection and allowing unauthenticated clients to access socket buffers, TCB's, and worker processes, making it easy to DoS the authentication mechanism itself. Ideally, a DDoS protection mechanism minimizes the resources consumed by unauthenticated clients. Kill-Bots introduces a modification to the server's TCP stack that can send a 1-2 packet puzzle at the end of the TCP handshake without maintaining any connection state, and while preserving TCP congestion control semantics.
- Kill-Bots improves performance, regardless of whether server overload is caused by DDoS attacks or true Flash Crowds, making it the **first system to address both DDoS and Flash Crowds within a single framework**. This is an important side effect of using admission control, which allows the server to admit new connections only if it can serve them.
- In addition, Kill-Bots requires no modifications to client software, is transparent to Web caches, and is robust to a wide variety of attacks (see §4).

We implement Kill-Bots in the Linux kernel and evaluate it in the wide-area network using PlanetLab. Additionally, we conduct an experiment on human users to quantify user willingness to solve graphical puzzles to

access a Web server. On a standard 2GHz Pentium IV machine with 1GB of memory and 512kB L2 cache running a mathopd [9] web-server on top of a modified Linux 2.4.10 kernel, Kill-Bots serves graphical tests in $31\mu s$; identifies malicious clients using the Bloom filter in less than $1\mu s$; and can survive DDoS attacks of up to 6000 HTTP requests per second without affecting response times. Compared to a server that does not use Kill-Bots, our system survives attack rates 2 orders of magnitude higher, while maintaining response times around their values with no attack. Furthermore, in our Flash Crowds experiments, Kill-Bots delivers almost twice as much goodput as the baseline server and improves response times by 2 orders of magnitude. These results are for an event driven OS that relies on interrupts. The per-packet cost of taking an interrupt is fairly large $\approx 10\mu s$ [23]. We expect better performance with polling drivers [30].

2 Threat Model

Kill-Bots aims to improve server performance under CyberSlam attacks, which mimic legitimate Web browsing behavior and consume higher layer server resources such as CPU, memory, database and disk bandwidth. Prior work proposes various filters for bandwidth floods [7, 16, 21, 27]; Kill-Bots does not address these attacks. Attacks on the server's DNS entry or on the routing entries are also outside the scope of this paper.

We assume the attacker may control an arbitrary number of machines that can be widely distributed across the Internet. The attacker may also have arbitrarily large CPU and memory resources. An attacker cannot sniff packets on the server's local network or on a major link that carries traffic for a large number of legitimate users. Further, the attacker does not have physical access to the server itself. Finally, the zombies cannot solve the graphical test and the attacker is not able to concentrate a large number of humans to continuously solve puzzles.

3 The Design of Kill-Bots

Kill-Bots is a kernel extension to Web servers. It combines authentication with admission control.

3.1 Authentication

During periods of severe overload, Kill-Bots authenticates clients before granting them service. The authentication has two stages that use different authentication mechanisms. Below, we explain in detail.

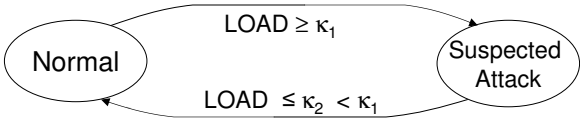


Figure 2: A Kill-Bots server transitions between NORMAL and SUSPECTED_ATTACK modes based on server load.

3.1.1 Activating the Authentication Mechanism

A Kill-Bots Web-server is in either of two modes, `NORMAL` or `SUSPECTED_ATTACK`, as shown in Fig. 2. When the Web server perceives resource depletion beyond an acceptable limit, κ_1, it shifts to the `SUSPECTED_ATTACK` mode. In this mode, every new connection has to solve a graphical test before allocation of any state on the server takes place. When the user correctly solves the test, the server grants the client access to the server for the duration of an HTTP session. Connections that began before the server switched to the `SUSPECTED_ATTACK` mode continue to be served normally until they terminate. However, the server will timeout these connections if they last longer than a certain duration (our implementation uses 5 minutes). The server continues to operate in the `SUSPECTED_ATTACK` mode until the load goes down to its normal range and crosses a particular threshold $\kappa_2 < \kappa_1$. The load is estimated using an exponential weighted average. The values of κ_1 and κ_2 will vary depending on the normal server load. For example, if the server is provisioned to work with 40% utilization, then one may choose $\kappa_1 = 70\%$ and $\kappa_2 = 50\%$.

A couple of points are worth noting. First, the server behavior is unchanged in the `NORMAL` mode, and thus the system has no overhead in the common case of no attack. Second, an attack that forces Kill-Bots to switch back-and-forth between the two modes is harmless because the cost for switching is minimal. The only potential switching cost is the need to timeout very long connections that started in the `NORMAL` mode. Long connections that started in a prior `SUSPECTED_ATTACK` mode need not be timed out because their users have already been authenticated.

3.1.2 Stage 1: CAPTCHA-Based Authentication

After switching to the `SUSPECTED_ATTACK` mode, the server enters $Stage_1$, in which it authenticates clients using graphical tests, i.e., CAPTCHAs [47], as in Fig. 4.

(a) Modifications to Server's TCP Stack: Upon the arrival of a new HTTP request, Kill-Bots sends a graphical test and validates the corresponding answer without allocating any TCBs, socket buffers, or worker processes on the server. We achieve this by a minor modification to the

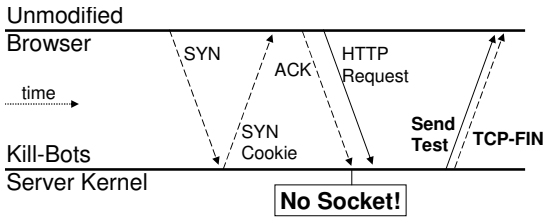


Figure 3: Kill-Bots modifies server's TCP stack to send tests to new clients without allocating a socket or other connection resources.

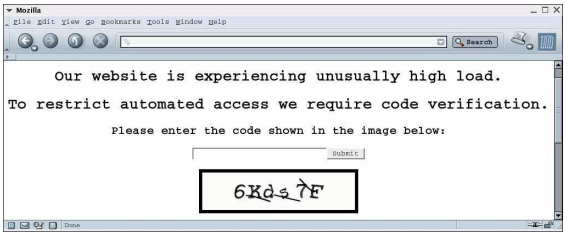


Figure 4: Screenshot of a graphical puzzle.

```
<html>
  <form method = "GET" action="/validate">
    <img src = "PUZZLE.gif">
    <input type = "password" name = "ANSWER">
    <input type = "hidden" name = "TOKEN" value = "[]">
  </form>
</html>
```

Figure 5: HTML source for the puzzle

Puzzle ID (P)	Random (R)	Creation Time (C)	Hash (P, R, C, secret)
32	96	32	32

Figure 6: Kill-Bots Token

server TCP stack. As shown in Fig. 3, similarly to a typical TCP connection, a Kill-Bots server responds to a SYN packet with a SYN cookie. The client receives the SYN cookie, increases its congestion window to two packets, transmits a SYNACKACK and the first data packet that usually contains the HTTP request. In contrast to a typical connection, the Kill-Bots kernel does not create a new socket upon completion of the TCP handshake. Instead, the SYNACKACK is discarded because the first data packet from the client repeats the same acknowledgment sequence number as the SYNACKACK.

When the server receives the client's data packet, it checks whether it is a puzzle answer. (An answer has an HTTP request of the form `GET /validate?answer=ANSWER`$_i$, where i is the puzzle id.) If the packet is not an answer, the server replies with a new graphical test, embedded in an HTML form (Fig. 5). Our implementation uses CAPTCHA images that fit in 1-2 packets. Then, the server immediately closes the connection by sending a FIN packet and does not wait for the FIN ack. On the other hand, the client packet could be a puzzle answer. When a human answers the graphical test, the HTML form (Fig. 5) generates an HTTP request `GET /validate?answer=ANSWER`$_i$ that reports the answer to the server. If the packet is an answer, the kernel checks the cryptographic validity of the ANSWER (see (c) below). If the check succeeds, a socket is established and the request is delivered to the application.

Note the above scheme preserves TCP congestion control semantics, does not require modifying the client software, and prevents attacks that hog TCBs and sockets by establishing connections that exchange no data.

(b) One Test Per Session: It would be inconvenient if legitimate users had to solve a puzzle for every HTTP request or every TCP connection. The Kill-Bots server gives an HTTP cookie to a user who solves the test correctly. This cookie allows the user to re-enter the system for a specific period of time, T (in our implementation, $T = 30$min). If a new HTTP request is accompanied by a cryptographically valid HTTP cookie, the Kill-Bots server creates a socket and hands the request to the application without serving a new graphical test.

(c) Cryptographic Support: When the Kill-Bots server issues a puzzle, it creates a Token as shown in Fig. 6. The token consists of a 32-bit puzzle ID P, a 96-bit random number R, the 32-bit creation time C of the token, and a 32-bit collision-resistant hash of P, R, and C along with the server secret. The token is embedded in the same HTML form as the puzzle (Fig. 6) and sent to the client.

When a user solves the puzzle, the browser reports the answer to the server along with the Kill-Bots token. The server first verifies the token by recomputing the hash. Second, the server checks the Kill-Bots token to ensure the token was created no longer than 4 minutes ago. Next, the server checks if the answer to the puzzle is correct. If all checks are successful, the server creates a Kill-Bots HTTP cookie and gives it to the user. The cookie is created from the token by updating the token creation time and recording the token in the table of valid Kill-Bots cookies. Subsequently, when a user issues a new TCP connection with an existing Kill-Bots cookie, the server validates the cookie by recomputing the hash and ensuring that the cookie has not expired, i.e., no more than 30 minutes have passed since cookie creation. The Kill-Bots server uses the cookie table to keep track of the number of simultaneous HTTP requests that belong to each cookie.

(d) Protecting Against Copy Attacks: What if the attacker solves a single graphical test and distributes the HTTP cookie to a large number of bots? Kill-Bots introduces a notion of per-cookie fairness to address this issue. Each correctly answered graphical test allows the client to execute a maximum of 8 simultaneous HTTP requests. Distributing the cookie to multiple zombies makes them compete among themselves for these 8 connections. Most legitimate Web browsers open no more than 8 simultaneous connections to a single server [20].

3.1.3 Stage 2: Authenticating Users Who Do Not Answer CAPTCHAs

An authentication mechanism that relies solely on CAPTCHAs has two disadvantages. First, the attacker can force the server to continuously send graphical tests, imposing an unnecessary overhead on the server. Second, and more important, humans who are unable or unwilling to solve CAPTCHAs may be denied service.

To deal with this issue, Kill-Bots distinguishes legitimate users from zombies by their reaction to the graphical test rather than their ability to solve it. Once the zombies are identified, they are blocked from using the server. When presented with a graphical test, legitimate users may react as follows: (1) they solve the test, immediately or after a few reloads; (2) they do not solve the test and give up on accessing the server for some period, which might happen immediately after receiving the test or after a few attempts to reload. The zombies have two options; (1) either imitate human users who cannot solve the test and leave the system after a few trials, in which case the attack has been subverted, or (2) keep sending requests though they cannot solve the test. However, by continuing to send requests without solving the test, the zombies become distinguishable from legitimate users, both human and machine.

In $Stage_1$, Kill-Bots tracks how often a particular IP address has failed to solve a puzzle. It maintains a Bloom filter [10] whose entries are 8-bit counters. Whenever a client is given a graphical puzzle, its IP address is hashed and the corresponding entries in the Bloom filter are incremented. In contrast, whenever a client comes back with a correct answer, the corresponding entries in the Bloom filter are decremented. Once all the counters corresponding to an IP address reach a particular threshold ξ (in our implementation ξ=32), the server drops all packets from that IP and gives no further tests to that client.

When the attack starts, the Bloom filter has no impact and users are authenticated using graphical puzzles. Yet, as the zombies receive more puzzles and do not answer them, their counters pile up. Once a client has ξ unanswered puzzles, it will be blocked. As more zombies get blocked, the server's load will decrease and approach its normal level. Once this happens the server no longer issues puzzles; instead it relies solely on the Bloom filter to block requests from the zombie clients. We call this mode $Stage_2$. Sometimes the attack rate is so high that even though the Bloom filter catches all attack packets, the overhead of receiving the packets by the device driver dominates. If the server notices that both the load is stable and the Bloom filter is not catching any new zombie IPs, then the server concludes that the Bloom filter has caught all attack IP addresses and switches off issuing puzzles, i.e., the server switches to $Stage_2$. If subsequently the load increases, then the server resumes issuing puzzles.

Var	Description
α	Admission Prob. Drop probability=1 $-\ \alpha$.
λ_a	Arrival rate of attacking HTTP requests
λ_s	Arrival rate of legitimate HTTP sessions
$\frac{1}{\mu_p}$	Mean time to serve a puzzle
$\frac{1}{\mu_h}$	Mean time to serve an HTTP request
ρ_p	Fraction of server time spent in authenticating clients
ρ_h	Fraction of server time spent in serving authenticated clients
ρ_i	Fraction of time the server is idle
$\frac{1}{q}$	Mean # of requests per legitimate session

Table 1: Variables used in the analysis

In our experiments, the Bloom filter detects and blocks all offending clients within a few minutes. In general, the higher the attack rate, the faster the Bloom filter will detect the zombies and block their requests. A full description of the Bloom filter is in §5. We detail Kill-Bots interaction with Web proxies/NATs in §8.

3.2 Admission Control

A Web site that performs authentication to protect itself from DDoS has to divide its resources between authenticating new clients and servicing those already authenticated. Devoting excess resources to authentication might leave the server unable to fully service the authenticated clients; thereby wasting the resources on authenticating new clients that it cannot serve. On the other hand, devoting excess resources to serving authenticated clients may cause the server to go idle because it hasn't authenticated enough new clients. Thus, there is an optimal authentication probability, α^*, that maximizes the server's goodput.

In [39], we have modeled a server that implements an authentication procedure in the interrupt handler. This is a standard location for packet filters and kernel firewalls [3, 29, 38]. It allows dropping unwanted packets as early as possible. Our model is fairly general and independent of how the authentication is performed. The server may be checking client certificates, verifying their passwords, or asking them to solve a puzzle. Furthermore, we make no assumptions about the distribution or independence of the inter-arrival times of legitimate sessions, or of attacker requests, or of service times.

The model in [39] computes the optimal probability with which new clients should be authenticated. Below, we summarize these results and discuss their implications. Table 1 describes our variables.

When a request from an unauthenticated client arrives, the server attempts to authenticate it with probability α and drop it with probability $1 - \alpha$. The optimal value of α–i.e., the value that maximizes the server's goodput (the CPU time spent on serving HTTP requests) is:

$$\alpha^* = \min\left(\frac{q\mu_p}{(B+q)\lambda_s + q\lambda_a}, 1\right), \text{ and } B = \frac{\mu_p}{\mu_h}, \quad (1)$$

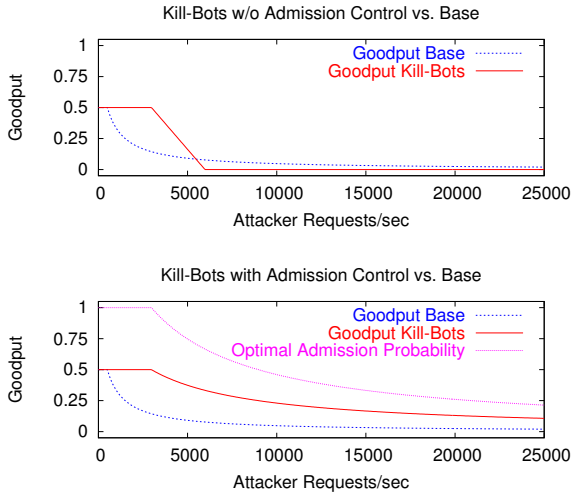


Figure 7: Comparison of the goodput of a base/unmodified server with a server that uses authentication only (TOP) and a server that uses both authentication & admission control (BOTTOM). Server load due to legitimate requests is 50%. The graphs show that authentication improves goodput, is even better with admission control, particularly at high attack rates.

where λ_a is the attack request rate, λ_s is the legitimate users' session rate, $\frac{1}{\mu_p}$ is the average time taken to serve a puzzle, $\frac{1}{\mu_h}$ is the average time to serve an HTTP request, and $\frac{1}{q}$ is the average number of requests in a session. This yields an optimal server goodput, which is given by:

$$\rho_g^* = \min\left(\frac{\lambda_s}{q\mu_h}, \frac{\lambda_s}{(1+\frac{q}{B})\lambda_s + q\frac{\lambda_a}{B}}\right). \quad (2)$$

In comparison, a server that does not use authentication has goodput:

$$\rho_g^b = \min\left(\frac{\lambda_s}{q\mu_h}, \frac{\lambda_s}{\lambda_s + q\lambda_a}\right). \quad (3)$$

To combat DDoS, authentication should consume fewer resources than service, i.e., $\mu_p >> \mu_h$. Hence, $B >> 1$, and the server with authentication can survive attack rates that are B times larger without loss in goodput.

Also, compare the optimal goodput, ρ_g^*, with the goodput of a server that implements authentication without admission control (i.e., $\alpha = 1$) given by:

$$\rho_g^a = \min\left(\frac{\lambda_s}{q\mu_h}, \max\left(0, 1 - \frac{\lambda_a + \lambda_s}{\mu_p}\right)\right). \quad (4)$$

For attack rates, $\lambda_a > \mu_p$, the goodput of the server with no admission goes to zero, whereas the goodput of the server that uses admission control decreases gracefully.

Fig. 7 illustrates the above results: A Pentium-IV, 2.0GHz 1GB RAM, machine serves 2-pkt puzzles at a peak rate of 6000/sec ($\mu_p = 6000$). Assume, conservatively, that each HTTP request fetches 15KB files ($\mu_h =$ 1000), that a user makes 20 requests in a session ($q = 1/20$) and that the normal server load is 50%. By substituting in Eqs. 3, 4, and 2, Fig. 7 compares the goodput of a server that does not use authentication (base server) with the goodput of a server with authentication only ($\alpha = 1$), and a server with both authentication and admission control ($\alpha = \alpha^*$). The top graph shows that authentication improves server goodput. The bottom graph shows the additional improvement from admission control.

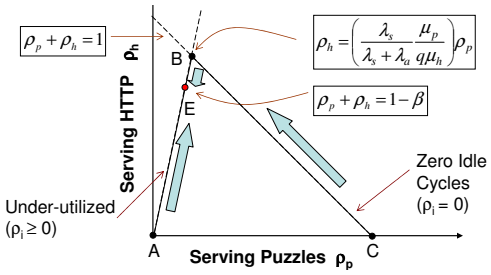


Figure 8: Phase plot showing how Kill-Bots adapts the admission probability to operate at a high goodput

3.3 Adaptive Admission Control

How to make the server function at the optimal admission probability? Computing α^* from Eq. 1 requires values for parameters that are typically unknown at the server and change over time, such as the attack rate, λ_a, the legitimate session rate, λ_s, and the number of requests per session, $\frac{1}{q}$.

To deal with the above difficulty, Kill-Bots uses an adaptive scheme. Based on simple measurements of the server's idle cycles, Kill-Bots adapts the authentication probability α to gradually approach α^*. Let ρ_i, ρ_p, ρ_h denote the fraction of time the server is idle, serving puzzles and serving HTTP requests respectively. We have:

$$\rho_h + \rho_p + \rho_i = 1. \quad (5)$$

If the current authentication probability $\alpha < \alpha^*$, the authenticated clients are too few and the server will spend a fraction of its time idle, i.e., $\rho_i > 0$. In contrast, if $\alpha > \alpha^*$, the server authenticates more clients than it can serve and $\rho_i = 0$. The optimal probability α^* occurs when the idle time transitions to zero. Thus, the controller should increase α when the server experiences a substantial idle time and decrease α otherwise.

However, the above adaptation rule is not as simple as it sounds. We use Fig. 8 to show the *relation* between the fraction of time spent on authenticating clients ρ_p and that spent serving HTTP requests ρ_h. The line labeled "Zero Idle Cycles" refers to the states in which the system is highly congested $\rho_i = 0 \rightarrow \rho_p + \rho_h = 1$. The line labeled "Underutilized" refers to the case in which the

system has some idle cycles, i.e., $\alpha < \alpha^*$. In this case, a fraction α of all arrivals are served puzzles. The average time to serve a puzzle is $\frac{1}{\mu_p}$. Thus, the fraction of time the server is serving puzzles $\rho_p = \alpha\frac{\lambda_s+\lambda_a}{\mu_p}$. Further, an α fraction of legitimate sessions have their HTTP requests served. Thus, the fraction of time the server serves HTTP is $\rho_h = \alpha\frac{\lambda_s}{q\mu_h}$, where $\frac{1}{\mu_h}$ is the per-request average service time, and $\frac{1}{q}$ is the average number of requests in a session. Consequently,

$$\forall \alpha < \alpha^* \ : \quad \rho_h = \left(\frac{\lambda_s}{\lambda_s+\lambda_a}\frac{\mu_p}{q\mu_h}\right)\rho_p,$$

which is the line labeled "Underutilized" in Fig. 8. As the fraction of time the system is idle ρ_i changes, the system state moves along the solid line segments A→B→C. Ideally, one would like to operate the system at point B which maximizes the system's goodput, $\rho_g = \rho_h$, and corresponds to $\alpha = \alpha^*$. However, it is difficult to operate at point B because the system cannot tell whether it is at B or not; all points on the segment B–C exhibit $\rho_i = 0$. It is easier to stabilize the system at point E where the system is slightly underutilized because small deviations from E exhibit a change in the value of ρ_i, which we can measure. We pick E such that the fraction of idle time at E is $\beta = \frac{1}{8}$.

Next, we would like to decide how aggressively to adapt α. Substituting the values of ρ_p and ρ_h from the previous paragraph in Eq. 5 yields:

$$\forall \alpha < \alpha^* : \quad \alpha\left(\frac{\lambda_a+\lambda_s}{\mu_p} + \frac{\lambda_s}{q\mu_h}\right) = 1-\rho_i.$$

Hence, $\forall \alpha[t], \alpha[t+\tau] < \alpha^*$:

$$\frac{\alpha[t+\tau]}{\alpha[t]} = \frac{1-\rho_i[t+\tau]}{1-\rho_i[t]} \ \Rightarrow \frac{\Delta\alpha}{\alpha[t]} = \frac{\Delta\rho_i}{1-\rho_i[t]},$$

where $\alpha[t]$ and $\alpha[t+\tau]$ correspond to the values at time t and τ seconds later. Thus, every τ=10s, we adapt the admission probability according to the following rules:

$$\Delta\alpha = \begin{cases} \gamma_1\alpha\frac{\rho_i-\beta}{1-\rho_i}, & \rho_i \geq \beta \\ -\gamma_2\alpha\frac{\beta-\rho_i}{1-\rho_i}, & 0 < \rho_i < \beta \\ -\gamma_3\alpha. & \rho_i = 0 \end{cases} \quad (6)$$

where γ_1, γ_2, and γ_3 are constants, which Kill-Bots set to $\frac{1}{8}$, $\frac{1}{4}$, and $\frac{1}{4}$ respectively. The above rules move α proportionally to how far the system is from the chosen equilibrium point E, unless there are no idle cycles. In this case, α is decreased aggressively to go back to the stable regime around point E.

4 Security Analysis

This section discusses Kill-Bots's ability to handle a variety of attacks from a determined adversary.

(a) Socially-engineered attack: In a socially-engineered attack, the adversary tricks a large number of humans to solving puzzles on his behalf. Recently, spammers employed this tactic to bypass graphical tests that Yahoo and Hotmail use to prevent automated creation of email accounts [4]. The spammers ran a porn site that downloaded CAPTCHAs from the Yahoo/Hotmail email creation Web page, forced its own visitors to solve these CAPTCHAs before granting access, and used these answers to create new email accounts.

Kill-Bots is much more resilient to socially engineered attacks. In contrast to email account creation where the client is given an ample amount of time to solve the puzzle, puzzles in Kill-Bots expire 4 minutes after they have been served. Thus, the attacker cannot accumulate a store of answers from human users to mount an attack. Indeed, the attacker needs a *continuous* stream of visitors to his site to be able to sustain a DDoS attack. Further, Kill-Bots maintains a loose form of fairness among authenticated clients, allowing each of them a maximum of 8 simultaneous connections. To grab most of the server's resources, an attacker needs to maintain the number of authenticated malicious clients much larger than that of legitimate users. For this, the attacker needs to control a server at least as popular as the victim Web server. Such a popular site is an asset. It is unlikely that the attacker will jeopardize his popular site to DDoS an equally or less popular Web site. Furthermore, one should keep in mind that security is a moving target; by forcing the attacker to resort to socially engineered attacks, we made the attack harder and the probability of being convicted higher.

(b) Polluting the Bloom filter: The attacker may try to spoof his IP address and pollute the Bloom filter, causing Kill-Bots to mistake legitimate users as malicious. This attack however is not possible because SYN cookies prevent IP spoofing and Bloom filter entries are modified *after* the SYN cookie check succeeds (Fig. 10).

(c) Copy attacks: In a copy attack, the adversary solves one graphical puzzle, obtains the corresponding HTTP cookie, and distributes it to many zombies to give them access to the Web site. It might seem that the best solution to this problem is to include a secure one-way hash of the IP address of the client in the cookie. Unfortunately, this approach does not deal well with proxies or mobile users. Kill-Bots protects against copy attacks by limiting the number of in-progress requests per puzzle answer. Our implementation sets this limit to 8.

(d) Replay attacks: A session cookie includes a secure hash of the time it was issued and is only valid during a certain time interval. If an adversary tries to replay a session cookie outside its time interval it gets rejected. An attacker may solve one puzzle and attempt to replay the "answer" packet to obtain many Kill-Bots cookies. Recall that when Kill-Bots issues a cookie for a valid an-

swer, the cookie is an updated form of the token (Fig 6). Hence, replaying the "answer" yields the same cookie.

(e) Database attack: The adversary might try to collect all possible puzzles and the corresponding answers. When a zombie receives a puzzle, it searches its database for the corresponding answer, and sends it back to the server. To protect from this attack, Kill-Bots uses a large number of puzzles and periodically replaces puzzles with a new set. Generation of the graphical puzzles is relatively easy [47]. Further, the space of all possible graphical puzzles is huge. Building a database of these puzzles and their answers, distributing this database to all zombies, and ensuring they can search it and obtain answers within 4 minutes (lifetime of a puzzle) is very difficult.

(f) Concerns regarding in-kernel HTTP header processing: Kill-Bots does not parse HTTP headers; it pattern matches the arguments to the `GET` and the `Cookie:` fields against the fixed string *validate* and against a 192-bit Kill-Bots cookie respectively. The pattern-matching is done in-place, i.e. without copying the packet and is cheap; $< 8\mu s$ per request (§6.1.2).

(g) Breaking the CAPTCHA: Prior work on automatically solving simple CAPTCHAs exists [33], but such programs are not available to the public for security reasons [33]. However, when one type of CAPTCHAs get broken, Kill-Bots can switch to a different kind.

5 Kill-Bots System Architecture

Fig. 9 illustrates the key components of Kill-Bots, which we briefly describe below.

(a) The Puzzle Manager consists of two components. First, a user-space stub that asynchronously generates new puzzles and notifies the kernel-space portion of the Puzzle Manager of their locations. Generation of the graphical puzzles is relatively easy [1], and can either be done on the server itself in periods of inactivity (at night) or on a different dedicated machine. Also puzzles may be purchased from a trusted third party. The second component is a kernel-thread that periodically loads new puzzles from disk into the in-memory Puzzle Table.

(b) The Request Filter (RF) processes every incoming TCP packet addressed to port 80. It is implemented in the bottom half of the interrupt handler to ensure that unwanted packets are dropped as early as possible.

Fig. 10 provides a flowchart representation of the RF code. When a TCP packet arrives for port 80, the RF first checks whether it belongs to an established connection in which case the packet is immediately queued in the socket's receive buffer and left to standard kernel processing. Otherwise the filter checks whether the packet starts a new connection (i.e., is it a SYN?), in which case, the RF replies with a SYNACK that contains a standard SYN cookie. If the packet is not a SYN, the RF examines

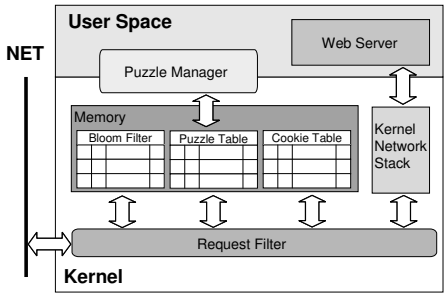


Figure 9: A Modular representation of the Kill-Bots code.

whether it contains any data; if not, the packet is dropped without further processing. Next, the RF performs two inexpensive tests in an attempt to drop unwanted packets quickly. It hashes the packet's source IP address and checks whether the corresponding entries in the Bloom filter have all exceeded ξ unsolved puzzles, in which case the packet is dropped. Otherwise, the RF checks that the acknowledgment number is a valid SYN cookie.

If the packet passes all of the above checks, the RF looks for 3 different possibilities: (1) this might be the first data packet from an unauthenticated client, and thus it goes through admission control and is dropped with probability $1 - \alpha$. If accepted, the RF sends a puzzle and terminates the connection immediately; (2) this might be from a client that has already received a puzzle and is coming back with an answer. In this case, the RF verifies the answer and assigns the client an HTTP cookie, which allows access to the server for a period of time; (3) it is from an authenticated client that has a Kill-Bots HTTP cookie and is coming back to retrieve more objects. If none of the above is true, the RF drops this packet. These checks are ordered according to their increasing cost to shed attackers as cheaply as possible.

(c) The Puzzle Table maintains the puzzles available to be served to users. To avoid races between writes and reads to the table, we divide the Puzzle Table into two memory regions, a write window and a read window. The Request Filter fetches puzzles from the read window, while the Puzzle Manager loads new puzzles into the write window periodically in the background. Once the Puzzle Manager loads a fresh window of puzzles, the read and write windows are swapped atomically.

(d) The Cookie Table maintains the number of concurrent connections for each HTTP cookie (limited to 8).

(e) The Bloom Filter counts unanswered puzzles for each IP address, allowing the Request Filter to block requests from IPs with more than ξ unsolved puzzles. Our implementation sets $\xi = 32$. Bloom filters are characterized by the number of counters N and the number of hash functions k that map keys onto counters. Our implementation uses $N = 2^{20}$ and $k = 2$. Since a potentially large set of keys (32-bit IPs), are mapped onto much smaller

storage (N counters), Bloom filters are essentially lossy. This means that there is a non-zero probability that all k counters corresponding to a legitimate user pile up to ξ due to collisions with zombies. Assuming a distinct zombies and uniformly random hash functions, the probability a legitimate client is classified as a zombie is approximately $(1 - e^{-ka/N})^k \approx (\frac{ka}{N})^k$. Given our choice of N and k, this probability for 75,000 zombies is 0.023.

6 Evaluation

We evaluate a Linux-based kernel implementation of Kill-Bots in the wide-area network using PlanetLab.

6.1 Experimental Environment

(a) Web Server: The web server is a 2GHz P4 with 1GB RAM and 512kB L2 cache running an unmodified mathopd [9] web-server on top of a modified Linux 2.4.10 kernel. We chose mathopd because of its simplicity. The Kill-Bots implementation consists of (1) 300 lines of modifications to kernel code, mostly in the TCP/IP protocol stack and (2) 500 additional lines for implementing the puzzle manager, the bloom filter and the adaptive controller. To obtain realistic server workloads, we replicate both static and dynamic content served by two web-sites, the CSAIL web-server and a Debian mirror.
(b) Modeling Request Arrivals: Legitimate clients generate requests by replaying HTTP traces collected at the CSAIL web-server and a Debian mirror. Multiple segments of the trace are played simultaneously to control the load generated by legitimate clients. A zombie issues requests at a desired rate by randomly picking a URI (static/dynamic) from the content available on the server.
(c) Experiment Setup: We evaluate Kill-Bots in the wide-area network using the setup in Fig. 11. The Web server is connected to a 100Mbps Ethernet. We launch CyberSlam attacks from 100 different nodes on PlanetLab using different port ranges to simulate multiple attackers per node. Each PlanetLab node simulates up to 256 zombies—a total of 25,600 attack clients. We emulate legitimate clients on machines connected over the Ethernet, to ensure that any difference in their performance is due to the service they receive from the Web server, rather than wide-area path variability.
(d) Emulating Clients: We use WebStone2.5 [2] to emulate both legitimate Web clients and attackers. WebStone is a benchmarking tool that issues HTTP requests to a web-server given a specific distribution over the requests. We extended WebStone in two ways. First, we added support for HTTP sessions, cookies, and for replaying requests from traces. Second, we need the clients to issue requests at specific rate independent of how the web-server responds to the load. For this, we rewrote WebStone's networking code using libasync [28], an asynchronous socket library.

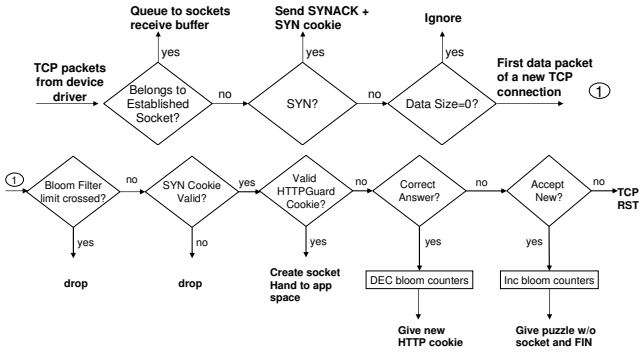


Figure 10: The path traversed by new sessions in Kill-Bots. This code-path is implemented by the Request Filter module.

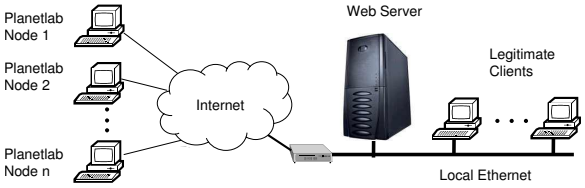


Figure 11: Our Experimental Setup.

Function	CPU Latency
Bloom Filter Access	.7 μs
Processing HTTP Header	8 μs
SYN Cookie Check	11 μs
Serving puzzle	31 μs

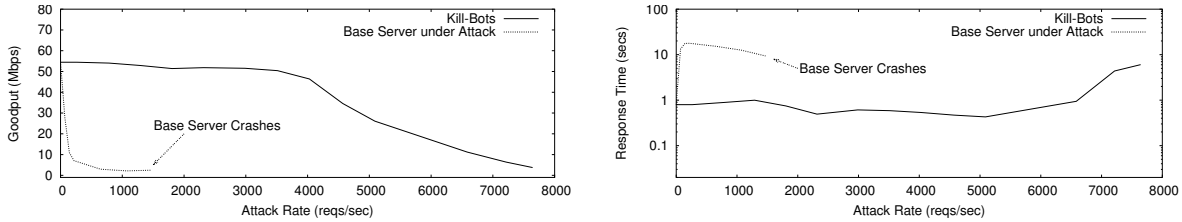
Table 2: Kill-Bots Microbenchmarks

6.1.1 Metrics

We evaluate Kill-Bots by comparing the performance of a base server (i.e., a server with no authentication) with its Kill-Bots mirror operating under the same conditions. Server performance is measured using these metrics:
(a) Goodput of legitimate clients: The number of bytes per second delivered to *all* legitimate client applications. Goodput ignores TCP retransmissions and is averaged over 30s windows.
(b) Response times of legitimate clients: The elapsed time before a request is completed or timed out. We timeout incomplete requests after 60s.
(c) Cumulative number of legitimate requests dropped: The total number of legitimate requests dropped since the beginning of the experiment.

6.1.2 Microbenchmarks

We run microbenchmarks on the Kill-Bots kernel to measure the time taken by the various modules. We use the x86 `rdtsc` instruction to obtain fine-grained timing information; `rdtsc` reads a hardware timestamp counter that is incremented once every CPU cycle. On our 2GHz



(a) Goodput (b) Response Time

Figure 12: Kill-Bots under CyberSlam: Goodput and average response time of legitimate users at different attack rates for both a base server and its Kill-Bots version. Kill-Bots substantially improves server performance at high attack rates.

web-server, this yields a resolution of 0.5 nanoseconds. The measurements are for CAPTCHAs of 1100 bytes.

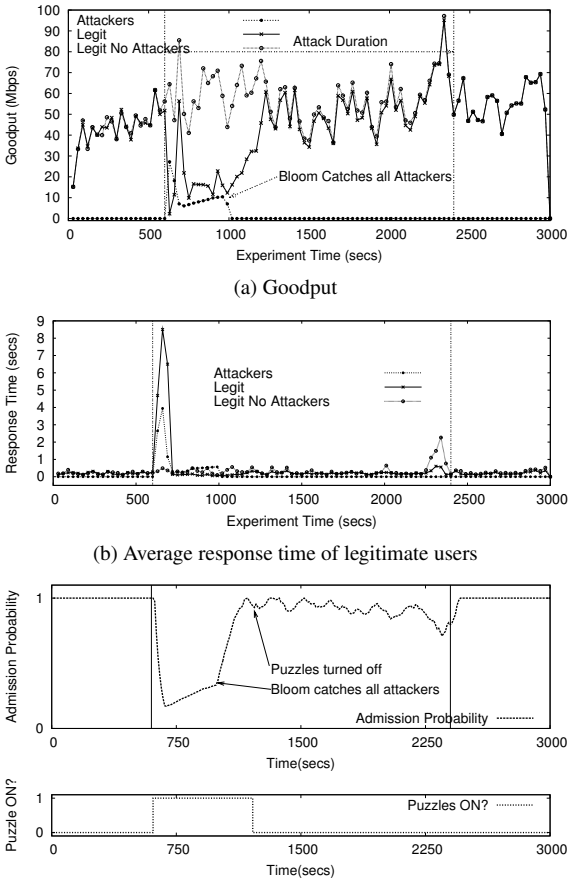
Table 2 shows our microbenchmarks. The overhead for issuing a graphical puzzle is $\approx 40\mu$s (process http header +serve puzzle), which means that the CPU can issue puzzles faster than the time to transmit a 1100B puzzle on our 100Mb/s Ethernet. However, the authentication cost is dominated by standard kernel code for processing incoming TCP packets, mainly the interrupts ($\approx 10\mu$s per packet [23], about 10 packets per TCP connection). Thus, the CPU is the bottleneck for authentication and as shown in §6.4, performing admission control based on CPU utilization is beneficial.

Note also that checking the Bloom filter is much cheaper than other operations including the SYN cookie check. Hence, for incoming requests, we perform the Bloom filter check before the SYN cookie check (Fig. 14). In $Stage_2$, the Bloom filter drops all zombie packets; hence performance is limited by the cost for interrupt processing and device driver access. We conjecture that using polling drivers [23, 30] will improve performance at high attack rates.

6.2 Kill-Bots under CyberSlam

We evaluate the performance of Kill-Bots under CyberSlam attacks, using the setting described in §6.1. We also assume only 60% of the legitimate clients solve the CAPTCHAs; the others are either unable or unwilling to solve them. This is supported by the results in §6.6.

Fig. 12 compares the performance of Kill-Bots with a base (i.e., unmodified) server, as the attack request rate increases. Fig. 12a shows the goodput of both servers. Each point on the graph is the average goodput of the server in the first twelve minutes after the beginning of the attack. A server protected by Kill-Bots endures attack rates multiple orders of magnitude higher than the base server. At very high attack rates, the goodput of the Kill-Bots server decreases as the cost of processing interrupts becomes excessive. Fig. 12b shows the response time of both web servers. The average response time experienced by legitimate users increases dramatically when the base server is under attack. In contrast, the average response time of users accessing a Kill-Bots server is unaffected by the ongoing attack.



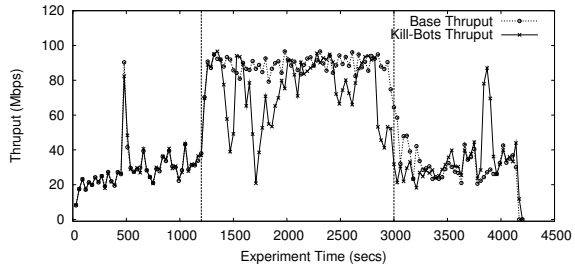
(a) Goodput

(b) Average response time of legitimate users

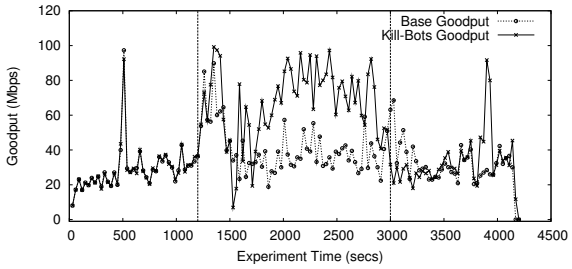
(c) Admission probability

Figure 13: Comparison of Kill-Bots' performance to server with no attackers when only 60% of the legitimate users solve puzzles. Attack lasts from 600s to 2400s. (a) Goodput quickly improves once bloom catches all attackers. (b) Response times improve as soon as the admission control reacts to the beginning of attack. (c) Admission control is useful both in $Stage_1$ and in $Stage_2$, after bloom catches all zombies. Puzzles are turned off when Kill-Bots enters $Stage_2$ improving goodput.

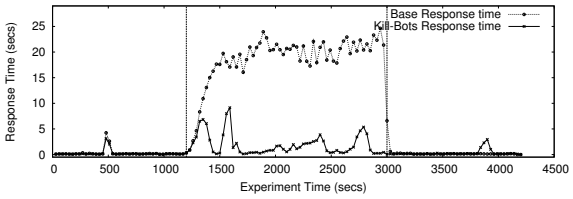
Fig. 13 shows the dynamics of Kill-Bots during a Cy-



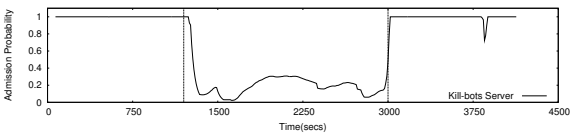
(a) Throughput of the server



(b) Average goodput of users



(c) Average response time perceived by users



(d) Admission probability at the server

Figure 14: Kill-Bots under Flash Crowds: The Flash Crowd event lasts from t=1200s to t=3000s. Though Kill-Bots has a slightly lower throughput, its Goodput is much higher and its avg. response time is lower.

berSlam attack, with $\lambda_a = 4000$ req/s. The figure also shows the goodput and mean response time with no attackers, as a reference. The attack begins at $t = 600$s and ends at $t = 2400$s. At the beginning of the attack, the goodput decreases (Fig. 13a) and the mean response time increases (Fig. 13b). Yet, quickly the admission probability decreases (Fig. 13c), causing the mean response time to go back to its value when there is no attack. The goodput however stays low because of the relatively high attack rate, and because many legitimate users do not answer puzzles. After a few minutes, the Bloom filter catches all zombie IPs, causing puzzles to no longer be issued (Fig. 13c). Kill-Bots now moves to $Stage_2$ and performs authentication based on just the Bloom filter. This causes a large increase in goodput (Fig. 13a) due to both the admission of users who were earlier unwilling or unable to solve CAPTCHAs and the reduction in authentication cost. In this experiment, despite the ongoing CyberSlam attack, Kill-Bots' performance in $Stage_2$ ($t = 1200s$ onwards), is close to that of a server not under attack. Note that the normal load significantly varies with time and the adaptive controller (Fig. 13c) reacts to this load $t \in [1200, 2400]s$, keeping response times low, yet providing reasonable goodput.

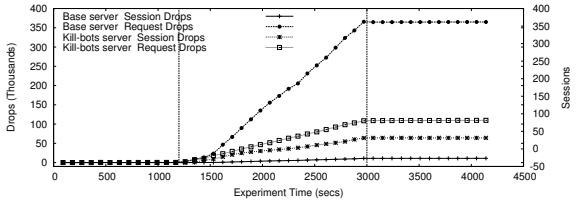


Figure 15: Cumulative numbers of dropped requests and dropped sessions under a Flash Crowd event lasting from $t = 1200s$ to $t = 3000s$. Kill-Bots adaptively drops sessions upon arrival, ensuring that accepted sessions obtain full service, i.e. have fewer requests dropped.

6.3 Kill-Bots under Flash Crowds

We evaluate the behavior of Kill-Bots under a Flash Crowd. We emulate a Flash Crowd by playing our Web logs at a high speed to generate an average request rate of 2000 req/s. The request rate when there is no flash crowd is 300 req/s. This matches Flash Crowd request rates reported in prior work [19, 20]. In our experiment, a Flash Crowd starts at $t = 1200s$ and ends at $t = 3000s$.

Fig. 14 compares the performance of the base server against its Kill-Bots mirror during the Flash Crowd event. The figure shows the dynamics as functions of time. Each point in each graph is an average measurement over a 30s interval. We first show the total throughput of both servers in Fig. 14a. Kill-Bots has slightly lower throughput for two reasons. First, Kill-Bots attempts to operate at β=12% idle cycles rather than at zero idle cycles. Second, Kill-Bots uses some of the bandwidth to serve puzzles. Fig. 14b reveals that the throughput figures are misleading; though Kill-Bots has a slightly lower throughput than the base server, its goodput is substantially higher (almost 100% more). This indicates that the base server wasted its throughput on retransmissions and incomplete transfers. Fig. 14c provides further supporting evidence– Kill-Bots drastically reduces the avg. response time.

That Kill-Bots improves server performance during Flash Crowds might look surprising. Although all clients in a Flash Crowd can answer the graphical puzzles, Kill-Bots computes an admission probability α such that the system only admits users it can serve. In contrast, a base server with no admission control accepts additional requests even when overloaded. Fig. 14d supports this argument by showing how the admission probability α changes during the Flash Crowd event to allow the server

to shed away the extra load.

Finally, Fig. 15 shows the cumulative number of dropped requests and dropped sessions during the Flash Crowd event for both the base server and the Kill-Bots server. Interestingly, the figure shows that Kill-Bots drops more sessions but fewer requests than the base server. The base server accepts new sessions more often than Kill-Bots but keeps dropping their requests. Kill-Bots drops sessions upon arrival, but once a session is admitted it is given a Kill-Bots cookie which allows it access to the server for 30min.

Note that Flash Crowds is just one example of a scenario in which Kill-Bots only needs to perform admission control. Kill-Bots can easily identify such scenarios–high server load but few bad bloom entries. Kill-Bots decouples authentication from admission control by no longer issuing puzzles; instead every user that passes the admission control check gets a Kill-Bots cookie.

6.4 Importance of Admission Control

In §3.2, using a simple model, we showed that authentication is not enough, and good performance requires admission control. Fig. 16 provides experimental evidence that confirms the analysis. The figure compares the goodput of a version of Kill-Bots that uses only puzzle-based authentication, with a version that uses both puzzle-based authentication and admission control. We turn off the Bloom filter in these experiments because we are interested in measuring the goodput gain obtained only from admission control. The results in this figure are fairly similar to those in Fig. 7; admission control dramatically increases server resilience and performance.

6.5 Impact of Different Attack Strategies

The attacker might try to increase the severity of the attack by prolonging the time until the Bloom filter has discovered all attack IPs and blocked them, i.e., by delaying transition from $Stage_1$ to $Stage_2$. To do so, the attacker uses the zombie IP addresses slowly, keeping fresh IPs for as long as possible. We show that the attacker does not gain much by doing so. Indeed, there is a tradeoff between using all zombie IPs quickly to create a severe attack for a short period vs. using them slowly to prolong a milder attack.

Fig. 17 shows the performance of Kill-Bots under two attack strategies; A fast strategy in which the attacker introduces a fresh zombie IP every 2.5 seconds, and a slow strategy in which the attacker introduces a fresh zombie IP every 5 seconds. In this experiment, the total number of zombies in the Botnet is 25000 machines, and the aggregate attack rate is constant and fixed at $\lambda_a = 4000$ req/s. The figure shows that the fast attack strategy causes

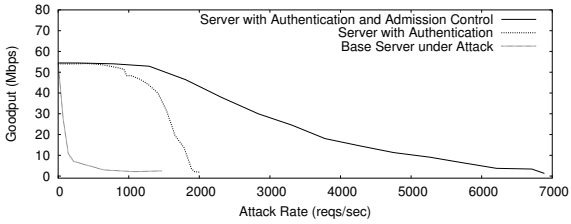
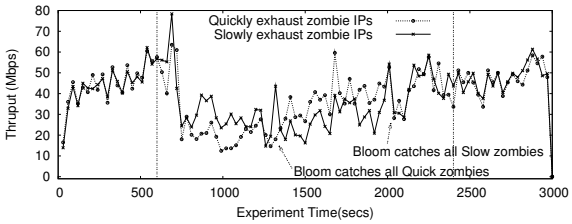
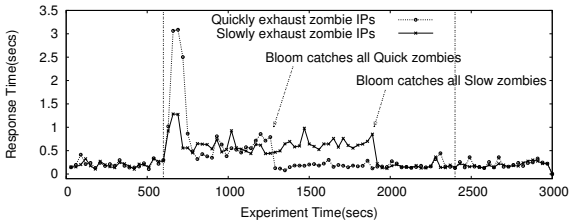


Figure 16: Server goodput substantially improves with adaptive admission control. Figure is similar to Fig. 7 but is based on wide-area experiments rather than analysis. (For clarity, the Bloom filter is turned off in this experiment.)



(a) Goodput



(b) Average response time of legitimate users

Figure 17: Comparison between 2 attack strategies; A fast strategy that uses all fresh zombie IPs in a short time, and a slow strategy that consumes fresh zombie IPs slowly. Graphs show a tradeoff; the slower the attacker consumes the IPs, the longer it takes the Bloom filter to detect all zombies. But the attack caused by the slower strategy though lasts longer has a milder impact on the goodput and response time.

a short but high spike in mean response time, and a substantial reduction in goodput that lasts for a short interval (about 13 minutes), until the Bloom filter catches the zombies. On the other hand, the slow strategy affects performance for a longer interval ($\sim$ 25 min) but has a milder impact on goodput and response time.

6.6 User Willingness to Solve Puzzles

We conducted a user study to evaluate the willingness of users to solve CAPTCHAs. We instrumented our research group's Web server to present puzzles to 50% of all external accesses to the *index.html* page. Clients that answer the puzzle correctly are given an HTTP cookie that allows them access to the server for an hour. The experiment lasted from Oct. 3 until Oct. 7. During that period, we registered a total of 973 accesses to the page, from 477 distinct IP addresses.

We compute two types of results. First, we filter out

Case	%Users
Answered puzzle	55%
Interested surfers who answered puzzle	74%

Table 3: The percentage of users who answered a graphical puzzle to access the Web server. We define interested surfers as those who access two or more pages on the Web site.

requests from known robots, using the `User-Agent` field, and compute the fraction of clients who answered our puzzles. We find that 55% of all clients answered the puzzles. It is likely that some of the remaining requests are also from robots but don't use well-known `User-Agent` identifiers, so this number underestimates the fraction of humans that answered the puzzles. Second, we distinguish between clients who check only the group's main page and leave the server, and those who follow one or more links. We call the latter *interested surfers*. We would like to check how many of the interested surfers answered the graphical puzzle because these users probably bring more value to the Web site. We find that 74% of interested users answer puzzles. Table 3 summarizes our results. These results may not be representative of users in the Internet, as the behavior of user populations may differ from one server to another.

7 Related Work

Related work falls into the following areas.

(a) Denial of Service: Much prior work on DDoS describes specific attacks (e.g., SYN flood [36], Smurf [11], reflector attacks [34] etc.), and presents detection techniques or countermeasures. In contrast to Kill-Bots, prior work focuses on lower layers attacks and bandwidth floods. The backscatter technique [31] detects DDoS sources by monitoring traffic to unused segments of the IP address space. Traceback [40] uses in-network support to trace offending packets to their source. Many variations to the traceback idea detect low-volume attacks [5, 41, 49]. Others detect bandwidth floods by mismatch in the volumes of traffic [16] and some [27] push-back filtering to throttle traffic closer to its source. Anderson et al. [7] propose that routers only forward packets with capabilities. Juels and Brainard [8] first proposed computational client puzzles as a SYN flood defense. Some recent work uses overlays as distributed firewalls [6, 21]. Clients can only access the server through the overlay nodes, which filter packets. The authors of [32] propose to use graphical tests in the overlay. Their work is different from ours because Kill-Bots uses CAPTCHAs only as an intermediate stage to identify the offending IPs. Further, Kill-Bots combines authentication with admission control and focusses on efficient kernel implementation.

(b) CAPTCHAs: Our authentication mechanism uses graphical tests or CAPTCHAs [47]. Several other reverse Turing tests exist [14, 22, 37]. CAPTCHAs are currently used by many online businesses (e.g. Yahoo!, Hotmail).

(c) Flash Crowds and Server Overload: Prior work [18, 48] shows that admission control improves server perfomance under overload. Some admission control schemes [15, 46] manage OS resources better. Others [20] persistently drop TCP SYN packets in routers to tackle Flash Crowds. Still others [42, 44] shed extra load onto an overlay or a peer-to-peer network. Kill-Bots couples admission control with authentication.

8 Limitations & Open Issues

A few limitations and open issues are worth discussing. First, Kill-Bots interacts in a complex manner with Web Proxies and NATs, which multiplex a single IP address among multiple users. If all clients behind the proxy are legitimate users, then sharing the IP address has no impact. In contrast, if a zombie shares the proxy IP with legitimate clients and uses the proxy to mount an attack on the Web server, Kill-Bots may block all subsequent requests from the proxy IP address. To ameliorate such fate-sharing, Kill-Bots increments the Bloom counter by 1 when giving out a puzzle but decrements the Bloom counters by $x \geq 1$ whenever a puzzle is answered. Kill-Bots picks x based on server policy. If $x > 1$, the proxy IP will be blocked only if the zombies traffic forwarded by the proxy/NAT is at least $x - 1$ times the legitimate traffic from the proxy. Further, the value of x can be adapted; if the server load is high even after the Bloom filter stops catching new IPs, Kill-Bots decreases the value of x because it can no longer afford to serve a proxy that has such a large number of zombies behind it.

Second, Kill-Bots has a few parameters that we have assigned values based on experience. For example, we set the Bloom filter threshold $\xi = 32$ because even legitimate users may drop puzzles due to congestion or indecisiveness and should not be punished. There is nothing special about 32, we only need a value that is neither too big nor too small. Similarly, we allow a client that answers a CAPTCHA a maximum of 8 parallel connections as a trade-off between the improved performance gained from parallel connections and the desire to limit the loss due to a compromised cookie.

Third, Kill-Bots assumes that the first data packet of the TCP connection will contain the `GET` and `Cookie` lines of the HTTP request. In general the request may span multiple packets, but we found this to happen rarely.

Forth, the Bloom filter needs to be flushed eventually since compromised zombies may turn into legitimate clients. The Bloom filter can be cleaned either by resetting all entries simultaneously or by decrementing the various entries at a particular rate. In the future, we will examine which of these two strategies is more suitable.

9 Conclusion

The Internet literature contains a large body of research on denial of service solutions. The vast majority assume that the destination can distinguish between malicious and legitimate traffic by performing simple checks on the content of packets, their headers, or their arrival rates. Yet, attackers are increasingly disguising their traffic by mimicking legitimate users access patterns, which allows them to defy traditional filters. This paper focuses on protecting Web servers from DDoS attacks that masquerade as Flash Crowds. Underlying our solution is the assumption that most online services value human surfers much more than automated accesses. We present a novel design that uses CAPTCHAs to distinguish the IP addresses of the attack machines from those of legitimate clients. In contrast to prior work on CAPTCHAs, our system allows legitimate users to access the attacked server even if they are unable or unwilling to solve graphical tests. We implemented our design in the Linux kernel and evaluated it in Planetlab.

Acknowledgements

We thank Shan Sinha for working on an early version of KillBots. We also thank Hari Balakrishnan, Robert Morris, Thomer Gil, David Andersen, Chuck Blake, Magdalena Balazinska, Allen Miu, Nate Kushman, our sheperd Stefan Savage, and the anonymous reviewers for their constructive comments. David Mazieres for libasync; Eddie Kohler for Click; and Michel Gorachzko for great technical support.

References

[1] Jcaptcha. jcaptcha.sourceforge.net/.

[2] Mindcraft Inc. Webstone - The Benchmark for Web Servers. http://www.mindcraft.com/webstone/.

[3] Netfilter/Iptables. http://www.netfilter.org.

[4] Porn Gets Spammers Past Hotmail, Yahoo Barriers. CNet News, May 2004. http://news.com.com/2100-1023_3-5207290.html.

[5] Alex Snoeren et al. Hash-Based IP Traceback. In *SIGCOMM*, 2001.

[6] D. Andersen. Mayday: Distributed Filtering for Internet services. In *USITS*, 2003.

[7] T. Anderson, T. Roscoe, and D. Wetherall. Preventing Internet Denial-of-Service with Capabilities. In *HotNets*, 2003.

[8] Ari Juels et al. Client Puzzles: A Cryptographic Countermeasure Against Connection Depletion Attacks. In *NDSS*, 1999.

[9] M. Boland. Mathopd. http://www.mathopd.org.

[10] A. Broder and M. Mitzenmacher. Network Applications of Bloom Filters: A Survey. In *Allerton*, 2002.

[11] CERT. Advisory CA-1998-01 Smurf IP Denial-of-Service Attacks, 1998. http://www.cert.org/advisories/CA-1998-01.html.

[12] CERT. Advisory CA-2003-20 W32/Blaster worm, 2003.

[13] CERT. Incident Note IN-2004-01 W32/Novarg.A Virus, 2004.

[14] A. Coates, H. Baird, and R. Fateman. Pessimal print: A Reverse Turing Test. In *IAPR*, 1999.

[15] G. Banga et al. Resource Containers: A New Facility for Resource Management in Server Systems. In *OSDI*, 1999.

[16] T. Gil and M. Poletto. MULTOPS: A Data-Structure for Bandwidth Attack Detection. In *USENIX Security*, 2001.

[17] E. Hellweg. When Bot Nets Attack. *MIT Technology Review*, September 2004.

[18] R. Iyer et al. Overload Control Mechanisms for Web Servers. In *Workshop on Perf. and QoS of Next Gen. Networks*, 2000.

[19] J. Jung et al. Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs. In *WWW*, 2002.

[20] H. Jamjoom and K. G. Shin. Persistent Dropping: An Efficient Control of Traffic. In *ACM SIGCOMM*, 2003.

[21] A. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure Overlay Services. In *ACM SIGCOMM*, 2002.

[22] G. Kochanski, D. Lopresti, and C. Shih. A Reverse Turing Test using speech. In *ICSLP*, 2002.

[23] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM TOCS*, 2000.

[24] J. Leyden. East European Gangs in Online Protection Racket, 2003. www.theregister.co.uk/2003/11/12/east_european_gangs_in_online/.

[25] J. Leyden. DDoSers attack DoubleClick, 2004. www.theregister.co.uk/2004/07/28/ddosers_attack_doubleclick/.

[26] J. Leyden. The Illicit trade in Compromised PCs, 2004. www.theregister.co.uk/2004/04/30/spam_biz/.

[27] R. Mahajan et al. Controlling High Bandwidth Aggregates in the Network. *CCR*, 2002.

[28] D. Mazieres. Toolkit for User-Level File Sys. In *USENIX*, 2001.

[29] S. McCanne. The Berkeley Packet Filter Man page, May 1991. BPF distribution available at ftp://ftp.ee.lbl.gov.

[30] J. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-driven Kernel. In *USENIX Tech. Conf.*, 1996.

[31] D. Moore, G. Voelker, and S. Savage. Inferring Internet Denial-of-Service Activity. In *USENIX Security*, 2001.

[32] W. G. Morein et al. Using Graphic Turing Tests to Counter Automated DDoS Attacks. In *ACM CCS*, 2003.

[33] G. Mori and J. Malik. Recognizing Objects in Adversarial Clutter: Breaking a Visual CAPTCHA. In *CVPR*, 2003.

[34] V. Paxson. An Analysis of Using Reflectors for Distributed Denial-of-Service Attacks. *ACM CCR*, 2001.

[35] K. Poulsen. FBI Busts Alleged DDoS Mafia, 2004. http://www.securityfocus.com/news/9411.

[36] L. Ricciulli, P. Lincoln, and P. Kakkar. TCP SYN Flooding Defense. In *CNDS*, 1999.

[37] Y. Rui and Z. Liu. ARTiFACIAL: Automated Reverse Turing Test Using FACIAL Features. In *Multimedia*, 2003.

[38] R. Russell. Linux IP Chains-HOWTO. http://people.netfilter.org/ rusty/ipchains/HOWTO.html.

[39] S. Kandula et al. Botz-4-sale: Surviving Organized DDoS Attacks That Mimic Flash Crowds. Technical Report TR-969, MIT., 2004.

[40] S. Savage et al. Practical Network Support for IP Traceback. In *SIGCOMM*, 2000.

[41] D. Song and A. Perrig. Advanced and Authenticated Marking Schemes for IP Traceback. In *INFOCOM*, 2001.

[42] T. Stading, P. Maniatis, and M. Baker. Peer-to-peer Caching Schemes to Address Flash Crowds. In *IPTPS*, 2002.

[43] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the Internet in Your Spare Time. In *USENIX Security*, 2002.

[44] A. Stavrou et al. A lightweight, robust, P2P system to handle Flash Crowds. *IEEE JSAC*, 2004.

[45] L. Taylor. Botnets and Botherds. http://sfbay-infragard.org.

[46] Thiemo Voigt et al. Handling Multiple Bottlenecks in Web Servers Using Adaptive Inbound Controls. In *High-Speed Networks*, 2002.

[47] L. von Ahn et al. Captcha: Using Hard AI Problems for Security. In *EUROCRYPT*, 2003.

[48] M. Welsh et al. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *SOSP*, 2001.

[49] A. Yaar et al. Pi: A Path Identification Mechanism to Defend against DDoS Attacks. In *IEEE Security & Privacy*, 2003.

Cashmere: Resilient Anonymous Routing

Li Zhuang, Feng Zhou
U. C. Berkeley
{zl,zf}@cs.berkeley.edu

Ben Y. Zhao
U. C. Santa Barbara
ravenben@cs.ucsb.edu

Antony Rowstron
Microsoft Research UK
antr@microsoft.com

Abstract

Anonymous routing protects user communication from identification by third-party observers. Existing anonymous routing layers utilize Chaum-Mixes for anonymity by relaying traffic through relay nodes called mixes. The source defines a static forwarding path through which traffic is relayed to the destination. The resulting path is fragile and shortlived: failure of one mix in the path breaks the forwarding path and results in data loss and jitter before a new path is constructed. In this paper, we propose Cashmere, a resilient anonymous routing layer built on a structured peer-to-peer overlay. Instead of single-node mixes, Cashmere selects regions in the overlay namespace as mixes. Any node in a region can act as the MIX, drastically reducing the probability of a mix failure. We analyze Cashmere's anonymity and measure its performance through simulation and measurements, and show that it maintains high anonymity while providing orders of magnitude improvement in resilience to network dynamics and node failures.

1 Introduction

In many applications it is desirable to hide the identity of the communicating parties from each other and third-party observers. The ability to anonymously route packets is used in many applications, such as anonymous web browsing [1], anonymous voting and in peer-to-peer applications wanting to ensure fair resource sharing [19].

The first-generation of applications that used anonymous routing, including the Anonymizer [1], were centralized, with central points of failure. More recent anonymous routing proposals [22, 30, 11] extend Chaum-Mixes [3] by forwarding traffic through a sequence of relays. Each relay is a single network endpoint. They attempt to ensure that the identity of the message source is never revealed to the destination, and the source and destination identities are hidden from relays and third-party observers. They achieve this by wrapping the payload and the sequence of relays through which a message is to be forwarded in layers of public key encryption, with one layer for each relay to be used. This requires that a set of relays be statically chosen at the beginning of a communication session. In general, if A sends a message M to B, then A defines a forwarding path that is a sequence of L relays $R_1, R_2, \ldots, R_L$. Each relay has a public/private key pair, where the public key of relay R_i is K_i. The message M is then sent encrypted in the form of $R_1 < R_2, < R_3, \ldots < R_L, < B, M >_{K_L} >_{K_{L-1}} \ldots >_{K_2} >_{K_1}$.

Successful end-to-end message delivery requires that every relay R_i in the forwarding path successfully decrypts its designated layer and forwards the message to the next relay. If the next relay has failed or is unreachable, then the message cannot be forwarded any further. When this occurs the source must discover the failure and then select a new set of live relays and resend the payload. Detecting failures in the routing path is made difficult because relays cannot send error messages to the anonymous source. This means that while these systems work in static and reliable networks, their performance degrades on less reliable wide-area links. They are also unlikely to function well on peer-to-peer and ad-hoc networks, where both end-point and link failure are observed regularly.

We propose a failure resilient anonymous routing system called Cashmere. Cashmere achieves resilience by using a set of distributed endpoints as a single virtual relay rather than a single endpoint. We refer to these endpoints as *relay groups*, and the forwarding path used in Cashmere is a sequence of *relay groups*. All members of a relay group share a public/private key pair. Layered encryption is still used on the forwarding path, using the public key of the relay group. Every member of the relay group has the ability to independently decrypt the next layer in the forwarding path. A forwarding path is valid as long as each relay group used in the forward-

ing path has at least one single live reachable member. While Chaum-Mixes route to the destination as the last hop, the destination in Cashmere is a member of any one of the relay groups on the forwarding path. The source randomly orders the relay groups to hide the destination relay group. When a message arrives at a member of a relay group, the receiver both anycasts the message to the next relay group and broadcasts the decrypted contents to all other members of the relay group. This ensures that if the destination is a member of the current group, it will receive the message.

Design Goals There are different types of anonymity [23]. Cashmere is designed to provide both *source anonymity* and *unlinkability* of source and destination. *Unlinkability* means that even if the source and destination can each be identified as participating in some communication, they can not be identified as communicating with each other. *Source anonymity* means that the identity of the source is hidden to all other nodes including the receiver. An attacker may be able to associate a set of messages with the same session but cannot determine the source, destination or the message payload. Provided the source does not divulge its identity in the message payload or collude with attackers, Cashmere provides both source anonymity and unlinkability even if the destination is controlled by an attacker. Cashmere can easily be extended to provide *destination anonymity*, where the destinations identity is hidden to all other nodes including the source, using an additional level of indirection.

Attack model We assume the attacker controls a fraction f of the nodes in the Cashmere network and these compromised nodes collude, sharing all information such as private keys. We assume a Byzantine failure model where compromised nodes can behave arbitrarily. The attacker can observe all messages sent over the network, regardless of whether the source or destination is controlled by the attacker, and there is zero latency for messages sent between compromised nodes.

The rest of this paper is structured as follows. We give an overview of related work and their limitations in Section 2. Next, we present the design of Cashmere in Section 3. We then discuss details of our current Cashmere implementation in Section 4. In Section 5, we analyze the level of anonymity in Cashmere and evaluate its security and performance using both simulation and measurements from an actual implementation. Finally, we outline future work and conclude in Section 6.

2 Related Works and Limitations

The original anonymous system redirected traffic through a centralized proxy [1]. Chaum [3] improved on this by using mix networks to create anonymous email, and inspired a number of subsequent systems [11, 24, 10, 7], including the Onion Routing system [22, 31]. Onion Routing relies on traffic redirection between a static set of dedicated onion routers that maintain pair-wise symmetric keys. To send a message, the source selects a set of currently active routers through which a message is forwarded. These requirements limit the scalability of Onion Routing, especially in environments with node churn. Tor [9] proposes using a directory server to maintain router information but this approach is also limited in scalability. It has also been shown that if the first or last router is compromised in an Onion Routing network, the source or destination is revealed [30].

Tarzan [11] also uses layered encryption and multi-hop routing. The source chooses a set of relays to act as a path and iteratively establishes a tunnel through these relays with symmetric keys between them. Hence, the creation of a tunnel incurs both significant computation overhead and delay. The tunnels are static and any relay failure requires formation of a new tunnel.

Crowds [23] and more recently AP3 [16] make use of probabilistic random forwarding. Crowds is limited in scalability because of its centralized admission control server, and has been shown to provide lower anonymity than Chaum-Mixes based systems [8].

Wright et al. [32, 33] have shown that relying on static forwarding paths impacts the anonymity properties of anonymous routing layers. They proposed a degradation attack applicable to Crowds, Onion Routing and other anonymizing systems that exploits the requirement to reconstruct the paths when they break due to node or link failure. During a long communication session, the path between source and destination is reconstructed many times, and each instance of the path must include the sender. After a large number of resets, the sender has much higher probability of being a path member than other nodes. Assume that the "first" attacker on each path (of the same session) logs its predecessor. After a number of path resets, the identity of the sender can be guessed with increasing probability.

Cashmere addresses these limitations by removing the reliance on static paths. By using flexible relay groups to maintain resilient long-lived paths, we improve performance by reducing path reconstruction time, and also reduce our vulnerability to the degradation attacks [32, 33] mentioned above. We gain these benefits with minimal loss to the level of anonymity attained compared to other Chaum-Mixes approaches.

3 Cashmere Architecture

Cashmere uses layered-encryption and multi-hop routing through relays. Instead of using a single node as a relay,

Cashmere uses a set of nodes that act as a virtual relay, called a *relay group*. All members of a relay group share a common public/private key pair.

A forwarding path consists of a sequence of relay groups. *Any* member of a relay group is able to decrypt the forwarding path information for a message and forward the message to the next relay group. The membership of the relay group can change dynamically. As long as the relay group has at least one member, it is able to successfully relay messages. This makes Cashmere extremely resilient to node churn. A relay group is an anycast group, and the forwarding of a message in analogous to anycasting to the next relay group. Unlike in Chaum-Mixes where the destination is the last hop, in Cashmere the destination is a member of one of the relay groups.

Cashmere is built on a structured overlay, and we leverage this to both dynamically create and maintain the relays groups as well as for routing between relay groups.

3.1 Structured Overlay Networks

Structured overlay networks provide a scalable routing substrate for building resilient, large-scale decentralized systems [21, 26, 29, 34]. A structured overlay is composed of a set of nodes, where each *node* represents an instance of a participant in the overlay. The structured overlay maintains a large *k-bit identifier space*, *e.g.* k=160. Nodes are assigned *nodeIDs* uniformly at random from this space, generated and signed by an off-line central authority (CA).

Most structured overlays support *Key-Based Routing* (KBR) [6], enabling applications to route a message to any specified key selected from the identifier space. These overlays dynamically map each key to a unique live node in the overlay, the *root* node for the key. The root could be the node with the nodeID numerically closest or with the longest prefix match to the key.

Each node in a structured overlay maintains a *routing table* that typically contains $O(\log N)$ nodeIDs and IP addresses of other nodes in the overlay, where N is the number of nodes in the overlay. By using nodeID constraints when choosing nodes for their routing table, they can route messages in $O(\log N)$ hops.

Cashmere is designed to use a prefix-routing based structured overlay, like Tapestry or Pastry. Routing in such overlays requires that at each hop the message is forwarded to a node whose nodeID shares a longer prefix match than the current node's nodeID. Figure 1 shows an example of prefix routing. At each hop the prefix match between the current nodeID and the key increases by one digit. These protocols are resilient to node churn [4], and can route around a large number of link failures [35].

Cashmere is being used as an anonymous routing infrastructure. The attacker could attempt to compromise the structured overlay, and thus compromise anonymity layer built on top it. To address this, we assume the structured overlay is secured against malicious nodes using the techniques described in [2] and [14]. In this paper we do not address the issue of denial of service (DoS) attacks. In Cashmere, DoS attacks affect performance but not the level of anonymity. Finally, our design can tolerate a large proportion of malicious nodes, and anonymity can be increased by creating longer relay paths even if a large proportion of the overlay has been corrupted. We also generate sufficient cover traffic[1] in the network to prevent simple traffic analysis attacks.

3.2 Relay groups

Relay group membership management in Cashmere exploits properties of the identifier space maintained by prefix-routing based structured overlays. In particular, for each k-bit nodeID there are k unique prefixes. For example, the 6-bit nodeID `101011` has prefixes: `1`, `10`, `101`, `1010`, `10101` and `101011`. In general, if there are N nodes it is expected that $N/2^m$ will share the same m-bit prefix.

In Cashmere, each relay group has a *groupID* which is an m-bit identifier, where $1 \leq m \leq k$. A node is a member of that relay group if the groupID is a prefix of its nodeID. Since nodeIDs are randomly assigned, nodes in a relay group are a random subset of the overlay nodes and exhibit independent failure patterns. Each prefix requires a public/private key pair and all nodes that share that prefix need both the public and private key. We assume these are generated and distributed using an off-line CA. In general, a user wishing to contribute a node to the system must obtain from the CA a signed k-bit nodeID and the set of k public/private keys associated with its nodeID and must have access to all the public keys of the other prefixes. Each nodeID must be unique, so the public/private key for the k-bit prefix will be unique to this nodeID.

The structured overlay routes messages between relay groups. The groupID is used as the key as a message is routed using KBR. As the message is routed, the first node that receives the message and shares the groupID prefix processes the message on behalf of the relay group. This node is referred to as the *relay group root*. Therefore, routing a message to a groupID is effectively performing an anycast to the relay group members.

Generally, if node A wants to route a message to node B anonymously, it selects a random sequence of m-bit groupIDs that defines the set of relay groups and includes the m-bit prefix of B. These are used to construct a forwarding path, *i.e.* a sequence of relay groups the message routes through. Since A selects the groupIDs randomly, the path cannot be predicted by others. The value of m

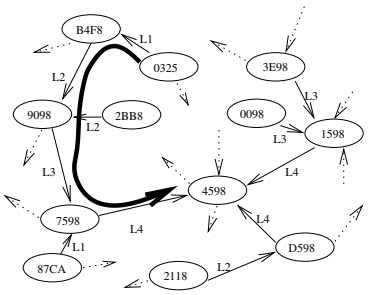


Figure 1: *Routing example in a structured overlay using prefix routing.* Node `5230` routes a message to the key `8954`. At each hop the message is forwarded to a node that shares a longer prefix with the key.

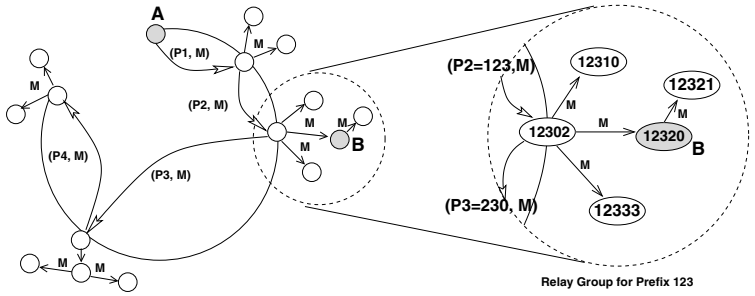


Figure 2: *A forwarding path from A to B composed of multiple relay groups.* Here a relay group is defined by a 3-digit prefix. At each relay group, the first node to receive the message broadcasts the message to all members of the group using *directed broadcast*. In the inset, node `12302` forwards the message to the rest of the relay group for prefix `123`.

controls the expected size of the relay group, and consequently the resilience against failures and malicious nodes. A encrypts the forwarding path in multiple layers using the public keys associated with each relay group. The overlay routes the message to the first relay group using its groupID. When any node matching the current prefix receives the message, it becomes the relay group root for that message, and uses the relay group's private key to decrypt the next layer of the path. This reveals the next groupID and the message is routed through the overlay to that prefix.

3.3 Decouple forwarding path and payload

Unlike other Chaum-Mixes based systems, Cashmere decouples the payload from the encrypted forwarding path, and encrypts the payload separately. This has the advantage that a source can reuse a forwarding path, avoiding multiple public key encryptions. The source caches the forwarding path, and only needs to perform a single public key encryption on each message using the destination's unique public key.

The source needs to encrypt each message payload such that it can only be decrypted by the true destination and such that each relay sees a different value for the payload (as do eavesdroppers). Suppose there are L relay groups in the forwarding path: $P_1, \cdots, P_L$ and the destination node B is in relay group P_d where $1 \leq d \leq L$. In order to encrypt the payload the source generates a symmetric key (R_i) for each relay group P_i, where $1 \leq i \leq L$. The source generates the payload:

$$\mathsf{Payload}_i = \begin{cases} \left\langle \mathsf{Payload}_{i+1} \right\rangle_{R_i} & 1 \leq i < d \\ \langle M \rangle_{\mathsf{PubKey}_B} & i = d \end{cases}$$

where $\langle M \rangle_{\mathsf{PubKey}_B}$ is the *real payload* encrypted by the destination's public key and $\langle \cdot \rangle_*$ indicates the content is encrypted using the key on the subscript. The source generates a forwarding path by:

$$\mathsf{Path}_i = \begin{cases} \langle \mathsf{Path}_{i+1}, P_{i+1}, R_i \rangle_{\mathsf{PubKey}_{P_i}} & 1 \leq i \leq L \\ \perp \text{ (termination)} & i = L+1 \end{cases}$$

The source then anycasts the tuple $[\mathsf{Path}_1, \mathsf{Payload}_1]$ to the first relay group P_1. In general, the i-th relay group root receives messages $[\mathsf{Path}_i, \mathsf{Payload}_i]$ from the previous relay group. The i-th relay group root uses the groups public key to decrypt the outer layer of Path_i, revealing Path_{i+1}, the identity of the next relay group, P_{i+1} and the symmetric key R_i. The i-th relay group root decrypts $\mathsf{Payload}_i$ using R_i, generating $\mathsf{Payload}_{i+1}$. Provided Path_{i+1} is not $\perp$ then the relay group root anycasts the tuple $[\mathsf{Path}_{i+1}, \mathsf{Payload}_{i+1}]$ to the next relay group P_{i+1}. During a single session, the source caches Path_1 and generates $\mathsf{Payload}_1$ for each message.

This process ensures that $\mathsf{Path}_i \neq \mathsf{Path}_j$ and $\mathsf{Payload}_i \neq \mathsf{Payload}_j$ if $i \neq j$. In particular, the source only encrypts the payload with the symmetric keys for the relay groups $R_1, \ldots, R_{d-1}$. The path has embedded within it the symmetric keys $R_1, \ldots, R_L$. At each of the relay groups $P_d, \ldots, P_L$ the payload will be decrypted using appropriate symmetric key, resulting in the forwarded payload being a random number. This ensures that $\mathsf{Payload}_i \neq \mathsf{Payload}_j$ if $i \neq j$.

However, there is no guarantee that when the message reaches P_d that the relay group root will be node B, as any member of a relay group can receive a message for its relay group. To ensure B receives the message, we multicast the payload to the entire relay group. If node X receives the message (thus becoming the relay group root for the message), then X decrypts the relay group's layer from the path in the message and decrypts the payload with the revealed R. X caches the map $\mathsf{Path}_i \leftrightarrow \langle \mathsf{Path}_{i+1}, P_{i+1}, R_i \rangle$ to reduce the computational load when further messages from the same ses-

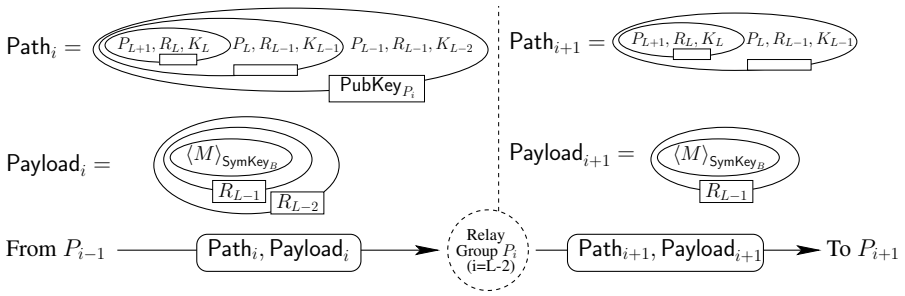


Figure 3: A detailed look at the path and payload components of a message, before and after processing at a relay group. The relay group root for P_i decrypts the layer around the path component to get P_{i+1}, R_i, K_i. It performs a symmetric decryption on the payload using R_i, and forwards the result to the relay group P_{i+1}.

sion are received. X forwards the message to the next relay group and broadcasts $\mathsf{Payload}_i$ to all members of the relay group (we discuss the exact mechanism in Section 4). No matter what position B's relay group is in the path, B will receive the message either directly or via a broadcast when the message routes to a member of its relay group. Only B will be able decrypt the payload successfully. An example of this Cashmere routing is shown in Figure 2.

The use of a broadcast has two implications addressed below; *(i)* that each node in a relay group has to perform an asymmetric decryption for each packet in a session; and *(ii)* malicious nodes can either drop messages or not broadcast them to the relay group. While such actions do not compromise anonymity they do negatively impact performance. We rely on end-to-end acknowledgments to detect failures and malicious nodes: if the source receives no acknowledgments, it can use timeouts to guide retransmission.

We eliminate the need to perform asymmetric encrypt/decrypt operations on the data payload by encrypting it using a symmetric key SymKey_B chosen when a source creates a path. In addition to the next relay group prefix, P_i, and a group session key, R_i, we embed another value K_i into the layered encrypted path. If destination B is in relay group d, then

$$K_d = \langle \mathsf{SymKey}_B | \mathsf{FLAG} \rangle_{\mathsf{PubKey}_B} ,$$

where | means concatenation. All other K_i values are random numbers. Now the format of Path_i is changed to

$$\mathsf{Path}_i = \langle \mathsf{Path}_{i+1}, P_{i+1}, R_i, K_i \rangle_{\mathsf{PubKey}_{P_i}} ,$$

and M is no longer encrypted with PubKey_B but is now encrypted with SymKey_B, $\langle M \rangle_{\mathsf{SymKey}_B}$. Figure 3 illustrates the full mechanism.

Now relay group roots broadcast $\langle K_i, \mathsf{Payload}_i \rangle$ to all members in the relay group. B decrypts K_d and identifies FLAG, thereby knowing that it is the destination. Using SymKey_B B can decrypt M. All other members of this relay group cache K_i. For future packets in the same session, they remember they are not the destination node and without further decryption operations. B caches SymKey_B and associates it with K_i and therefore only needs to perform symmetric decryption for subsequent session payloads.

This also has the advantage that relay group roots can cache $\langle \mathsf{Path}_{i+1}, P_{i+1}, R_i, K_i \rangle$ if they have already forwarded messages for the same session. Relay group roots can identify messages using Path_i as the session ID, hence no asymmetric decryption is necessary.

3.4 Anonymous Reply Addresses

A destination can reply to a source without sacrificing source anonymity or requiring state to be stored in the relay groups in the forwarding path. The destination can reply to the source either a pre-formatted reply message (*e.g.* an acknowledgment) or a message containing an arbitrary payload. The reply message shares all of the performance and security benefits with the anonymous messages from source to destination.

Node A wishes to send an anonymous message to B and receive a reply. A creates a forwarding path to B as described, but also generates a return forwarding path from B to A. A does this by randomly selecting L relay groups $(P'_1, \ldots, P'_L)$. The set of relay groups used in the return forwarding path may or may not intersect with the set of relay groups used in the forwarding path from A to B. A ensures that a relay group containing itself, $P_{d'}$, is included in the return path. A sends ReplyAddrInfo as part of the payload to B, where:

$$\mathsf{ReplyAddrInfo} = \left\langle \mathsf{Path}'_1, P'_1, \mathsf{SymKey}_A \right\rangle$$

$$\mathsf{Path}'_i = \begin{cases} \langle \mathsf{Path}'_{i+1}, P'_{i+1}, R'_i, K'_i \rangle_{\mathsf{PubKey}_{P'_i}} & 1 \le i \le L \\ \perp \text{ (termination)} & i = L+1 \end{cases}$$

$$K'_i = \begin{cases} k'_i & i \neq d' \\ \langle \mathsf{SymKey}_A | \mathsf{FLAG} \rangle_{\mathsf{PubKey}_A} & i = d' \end{cases}$$

where k'_i and R'_i are selected uniformly at random, $P'_{L+1} = \perp$. If B wants to send a payload M' to A, it sends Msg' as $[\mathsf{Path}'_1, \langle M' \rangle_{\mathsf{SymKey}_A}]$ to P'_1. While Msg' is created by B, it knows nothing about the path and the source. The root of each relay group P'_i decrypts Path'_i the same as in the forwarding path, while it encrypts $\mathsf{Payload}'_i$ with R'_i to get $\mathsf{Payload}'_{i+1} = \langle \mathsf{Payload}'_i \rangle_{R'_i}$. Node A who is located in relay group $P'_{d'}$ will receive message $\langle K'_{d'}, \mathsf{Payload}'_{d'} \rangle$, where $\mathsf{Payload}'_{d'}$ is the layered encryption of $\langle M \rangle_{\mathsf{SymKey}_A}$ by $R'_1, \ldots, R'_{d'-1}$. After A decrypts $K'_{d'}$ using PubKey_A, A can use SymKey_A to identify which session the reply belongs to, and thus the keys R'_i $(1 \leq i < d)$ to decrypt $\mathsf{Payload}'_{d'}$. All caching schemes used in the forwarding path also apply to the return path.

3.5 Selection of GroupID and Path Length

The final issue is how a source selects groupIDs for relay groups. Observation 1 shows the relation between the length of groupIDs and relay group sizes.

OBSERVATION 1: **(Distribution of Relay Group Sizes)** *Let N be the number of nodes in the overlay and nodeIDs are assigned to all nodes uniformly at random. The size of relay groups defined by a m-digit groupID is Poisson distributed with parameter $\rho = \frac{N}{2^m}$. The expected size of the relay group is ρ. [Proof omitted]*

A valid groupID requires that there exists at least one node that has the groupID as a prefix. As N is much smaller than the size of the nodeID identifier space, there will be many invalid groupIDs. From Observation 1, the probability that a groupID is valid is $p_1 = 1 - e^{-\rho}$. When a node forms a path by selecting groupIDs uniformly at random, the chance that the path contains only valid groupIDs is $(p_1)^L = (1 - e^{-\rho})^L$, where L is the number of relay groups used in the path. The expected number of tries to generate a valid path, one that is composed on only valid groupIDs, is $\frac{1}{(1-e^{-\rho})^L}$. Table 1 shows the average number of tries to generate a valid path is slightly larger than 1 under typical L and ρ values.

In Cashmere, nodes independently (without external communication) select per-session values of m (which determines ρ) and L to control tradeoffs between churn resilience, anonymity and overhead. We discuss this in Section 5.1. In general, choosing a value of between 3 and 5 for ρ, and a value of L between 4 and 8 provides a good combination of efficiency and resiliency. Because nodeIDs are uniformly distributed, nodes can locally estimate N using their routing tables. From Observation 1, a node can always get the average relay group size (ρ) it wants by selecting a proper prefix length m. The design of Cashmere removes the high cost of maintaining complete or near-complete overlay membership information.

	$L=4$	$L=5$	$L=6$	$L=7$	$L=8$
$\rho=4$	1.0767	1.0968	1.1173	1.1381	1.1594
$\rho=5$	1.0274	1.0344	1.0414	1.0485	1.0556

Table 1: Average number of tries to get a valid path.

4 Implementation

We implemented Cashmere on top of FreePastry [12], a Java implementation of Pastry [26]. The implementation uses RSA (with 512-bit key length) and Blowfish (with 128-bit key length) as the asymmetric key and symmetric key ciphers, and uses the Cryptix [5] crypto library.

Applications use a simple Cashmere API. The source creates an *AnonymousChannel* object specifying a destination nodeID, and uses it to forward payloads. An application instance running on the destination node receives an up-call with the payload.

The Cashmere implementation ensures that relay group roots cache Path_i information and all nodes cache K_i as described in the previous section.

Our implementation performs relay group broadcast of $\langle K_i, \mathsf{Payload}_i \rangle$ using the leaf sets that are maintained by each node [26]. The leaf set is a set of pointers to the immediate l neighbors in the identifier space, where typical $l = 8$. If the leaf set does not contain all members of the relay group, nodes on the edge of the leaf set forward the message to their leaf set members. This recursive process continues until all members of the relay group have received the message.

One practical issue in the encoding of the path is that it is desirable for it to have the same length all along the forwarding path. This way no information about the route can be obtained by simply observing the size changes of the path onion. Previous work discussed these length-preserving Chaum-Mixes. A simple scheme is implemented in Mixmaster [18], and [17] presents a more sophisticated, provably secure scheme. Our prototype currently uses the basic layered encryption, and thus the path size decreases after each relay group. Changing the encoding scheme to preserve message length is straightforward and orthogonal to the design and performance of the overall system.

5 Analysis and Evaluation

5.1 Anonymity Measurement

We analyze two types of anonymity provided by Cashmere: source anonymity and unlinkability of source and destination. We quantify Cashmere's anonymity parameterized by:

- N: network size;

- f: fraction of malicious nodes in the network;
- ρ: average relay group size ($\rho = N/2^m$);
- L: number of relay groups in a path, the path length.

The parameter f has two implications: *(i)* the probability that compromised nodes are on the relay path; and *(ii)* the fraction of relay group private keys known by the attacker. For each compromised nodeID the attacker will know the relay group private keys for all prefixes associated with the nodeID. The probability that the attacker knows a m-bit prefix private key is $p_2 = 1 - e^{f\rho}$. The attacker can obtain prefix private keys either by compromising other nodes or through obtaining nodeIDs from the CA. We assume prefix private keys are leaking slowly, and the offline CA can slowly issue new prefix keys and revoke prior prefix keys over time. If the attacker knows the private key for a relay group we refer to the relay group as being compromised.

Our anonymity measurement follows the anonymity definition by Pfitzmann et al. [20]: "Anonymity is the state of being not identifiable within a set of subjects, the *anonymity set*." In a network with a finite set Ω ($|\Omega| = N$) of nodes, *ideal anonymity* is achieved when all nodes look equally likely to be the source or destination to an attacker, *e.g.* the anonymity set is Ω. In reality, based on information leaked from the system, some nodes look more likely to be the source or destination than others. That is, the attacker knows that the source (or destination) is in Ω_i with probability $\Pr(\Omega_i)$ [2], where $\Omega = \bigcup_i \Omega_i$. For example, the *worst anonymity* is the attacker identifies the source or destination as u_0; $\{u_0\}$ is assigned with probability 1 and $\Omega \backslash \{u_0\}$ with probability 0. We use the metric proposed in [8, 28] to measure the anonymity of our design as a proportion of the ideal entropy achievable in a given network. We briefly describe the entropy-based metric as follows:

DEFINITION 5.1. **(Entropy of a System).** *Ω is the (finite) set of all nodes in the network. Using knowledge of leaked information from the system, an attacker assigns each node u ($u \in \Omega$) a probability p_u as being the source or destination of a message. System entropy is defined as:*

$$H(\Omega) = -\sum_{u \in \Omega} p_u \log_2(p_u).$$

If we have ideal anonymity, all nodes look equal to attackers: $\forall u \in \Omega$, $p_u = \frac{1}{|\Omega|}$. The entropy of ideal anonymity is $H_m(\Omega) = \log_2(|\Omega|)$, which is the maximum entropy achieved in a network of $|\Omega|$ nodes.

DEFINITION 5.2. **(Anonymity of a System).** *The anonymity of a system is measured as:*

$$\frac{H(\Omega)}{H_m(\Omega)} = \frac{-\sum_{u \in \Omega} p_u \log_2(p_u)}{\log_2(|\Omega|)}.$$

Definition 5.2 shows that the anonymity of a system is measured by the real entropy of a system over the maximum (*i.e.* ideal anonymity) entropy the system could achieve: $0 \leq \frac{H(\Omega)}{H_m(\Omega)} \leq 1$.

The entropy definition above is more precise than the straightforward probability definition of *the probability that the attacker knows the sender or receiver*. For example, let us consider source anonymity in network of 10000 nodes. In an anonymity system AS_1 with attacker T:

- T discovers the source of 5% of messages;
- T can limit sources of 40% of messages to a small subset of nodes, e.g. 100;
- For the other 55% of messages, all nodes look equally likely to be a source to T.

In another anonymity system AS_2,

- T discovers the source of 5% of messages;
- For the other 95% of messages, all nodes look equally likely to be a source to T.

Using the probability that T knows the sender or receiver, both AS_1 and AS_2 have anonymity of 0.95. Using the entropy definition, the anonymity of AS_1 is:

$$0.05*0+0.40*\frac{-100 * \log_2(\frac{1}{100})}{-10000 * \log_2(\frac{1}{10000})}+0.55*1 = 0.552;$$

and anonymity of AS_2 is: $0.05 * 0 + 0.95 * 1 = 0.95$.

The entropy definition is more precise, capturing that AS_2 provides better anonymity. In AS_1 the attacker knows more information about the sources than in AS_2.

The anonymity of Cashmere is determined by ρ and L given the fraction of compromised nodes. Anonymity increases with larger values of L. Intuitively, the destination is hidden among all relay group members and ρ and L determine the number of nodes in all relay groups. However, as ρ increases, which means a shorter prefix is selected for groupID and the attacker has more chance to know consecutive relay groups, the anonymity decreases. Larger ρ also means more resilience and a higher relay group broadcast overhead. From analysis and experimentation, good typical values for ρ are between 3 to 5. In this section, we perform simulations with a network of $16,384$ nodes. GroupIDs have a prefix length of 12 bits, such that the expected size of relay groups $\rho = 4$ nodes. We compute unlinkability and source anonymity using the entropy definition. We first assume that attackers only see their own traffic, and simulate unlinkability and source anonymity given different parameters of (f, L). We then analyze the security of Cashmere against traffic analysis attacks.

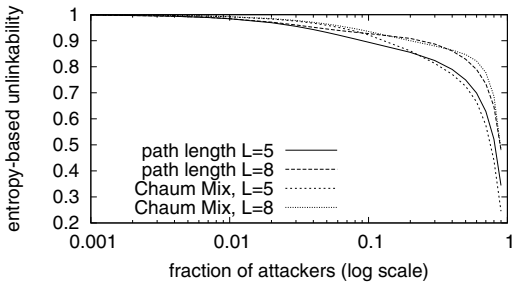


Figure 4: Anonymity measurement of unlinkability.

5.1.1 Unlinkability

In our simulations, the attacker gathers information observed from compromised nodes and maintains, for each pair of nodes (u_i, u_j), a probability p_{ij} that the pair are a source and destination.

Using the entropy definition, we can measure unlinkability using the relative entropy to ideal unlinkability:

$$\frac{\sum p_{ij}\log_2(p_{ij})}{N^2\cdot\left(\frac{1}{N^2}\log_2(\frac{1}{N^2})\right)} = \frac{-\sum p_{ij}\log_2(p_{ij})}{2\log_2 N}.$$

If the attacker believes u_i is the source with probability p_i and u_j is the destination with probability p_j, then $p_{ij} = p_i p_j$.

We assume the attacker determines the exact number of relay groups L used for a message [3]. We also assume the attacker knows a chain of n consecutive relay groups on the path of a message, each containing ρ_i nodes. Assuming there is enough cover traffic, the attacker cannot attribute discrete chains in the path to the same session, because the path onion and the observed payload are completely different at each relay group. Therefore, the attacker's knowledge about a message only comes from one consecutive chain on the relay path.

The source is indistinguishable from the *relay group root* of the immediately preceding relay group. When the first relay group root on the chain is non-malicious and known by the attacker, the attacker infers that the source is the first root with probability $\frac{1}{L-n+1}$ and the source is among all other non-malicious nodes with probability $1-\frac{1}{L-n+1}$. That is, for each non-malicious node u, the attacker assigns probability of u being the source as:

$$p_u = \begin{cases} \frac{1}{L-n+1} & \text{the first relay group root on the chain} \\ \left(1-\frac{1}{L-n+1}\right)\cdot\frac{1}{(1-f)N-1} & \text{otherwise} \end{cases}$$

When the first root on the chain is not known by the attacker or is malicious itself, all non-malicious nodes look equally to be the source, each with probability $\frac{1}{(1-f)N}$.

Let S be the set of nodes that are in the chain of relay groups known by the attacker, $|S| = \sum_{i=1}^{n}\rho_i$. The set of S is composed of both a set S_1 of malicious nodes and a set S_2 of non-malicious nodes: $S = S_1 \cup S_2$, $|S_1| = f|S|$ and $|S_2| = (1-f)|S|$. The expected number of nodes in all relay groups is $L\rho$. If the destination is among S_1, the attacker knows the destination and unlinkability becomes the source anonymity problem that we discuss in Section 5.1.2. If the destination is among S_2, the attacker infers that each node in S_2 is the destination with probability $\frac{1}{L\rho - f|S|}$ and the destination is among other nodes outside S with probability $1-\frac{(1-f)|S|}{L\rho-f|S|}$. That is, for each node u not in S_1, the attacker assigns the probability of u being the destination as:

$$p_u = \begin{cases} \frac{1}{L\rho-f|S|} & u \in S_2 \\ \left(1-\frac{(1-f)|S|}{L\rho-f|S|}\right)\cdot\frac{1}{N-|S|} & u \in \bar{S} \end{cases}$$

The number of relay groups compromised (*i.e.* n) is closely related to the fraction f of compromised nodes. If the compromised node was not the relay group root then the attacker would only learn the value of K_i and the payload, which is broadcast to the relay group. When the compromised node is the relay group root for a message the attacker also discovers the identity of the next relay group. If the compromised node is on the intermediate overlay hops between two relay group the attacker knows the previous or/and the next relay group root.

In Figure 4, we compare through simulation Cashmere's unlinkability metric to that of Chaum-Mixes approaches under different parameters of (L, f), ignoring eavesdropping and traffic analysis (see Section 5.1.3). In the simulation, we setup a relay path of length L, assign each node on the path and in the relay groups as compromised consistent with parameter f, count the probability of different cases that the attacker knows n consecutive groups, and compute the entropy in all cases. Then the entropy of the system is the average over all cases [8, 28].

The results show that Cashmere has similar anonymity to Chaum-Mixes. Cashmere even behaves better than Chaum-Mixes for small L and f near 1, when the whole Chaum-Mixes path is controlled by attackers with high probability while Cashmere still benefits from the anonymity among relay group members. We also measured how the level of unlinkability varies with network size and, as expected, unlinkability is largely independent of network size. Increasing network size from 20K nodes to 2 million nodes results in less than 3% variation in unlinkability. Reducing the network size to 64 provides similar unlinkability under the same f as large networks as long as ρ and L are set the same. Thus, bootstrapping Cashmere requires a small initial network of trusted nodes and then other nodes can join the network while maintaining the fraction of malicious nodes in the network as f.

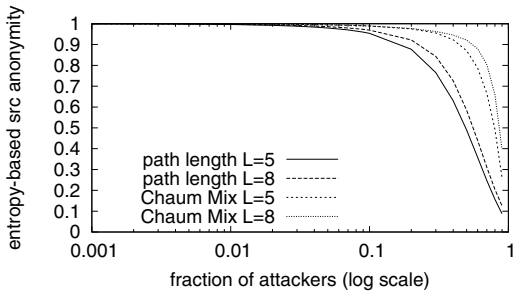


Figure 5: Source anonymity with anonymous messages.

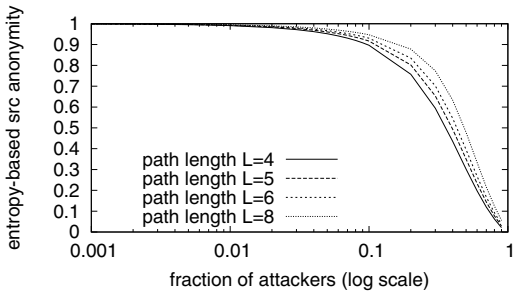


Figure 6: Source anonymity with anonymous channels.

5.1.2 Source Anonymity

In source anonymity, the destination colludes with other malicious nodes to find the source's identity. If Cashmere is being used for one-way communication (anonymous message) the attacker infers the first relay group root on the chain (which includes the destination's relay group) as the source with probability $\frac{1}{L-n+1}$ if the first root is non-malicious, where n is the length of the chain. Figure 5 compares the source anonymity of Cashmere with Chaum-Mixes, assuming no traffic analysis attacks, for one-way communication. We see that like Chaum-Mixes, Cashmere has high source anonymity when $f < 0.3$ and increasing L improves anonymity.

If Cashmere is being used for two-way communication (anonymous channel), the attacker has two ways to discover the source; *(i)* discover the first relay group root on the chain of consecutive relay groups which includes the destination's relay group (as for one-way), or *(ii)* the attacker compromises consecutive relay groups used on the return path from the destination to the source. Even if the attacker compromises all L of the return path relay groups, the attacker only knows the source is a member of one of these relay groups (the probability is the same as in Section 5.1.1).

Figure 6 shows the results for anonymous channels. The results show anonymous channels provide lower anonymity compared to anonymous messages due to the vulnerability of the return path. Finally, we also analyzed the impact of network size on source anonymity and, as before, increasing or decreasing the network size had negligible impact.

5.1.3 Robustness against Traffic Analysis

Our previous simulations disregarded the impact of traffic analysis. In practice, however, attackers may monitor part or all of the network traffic and use patterns to trace session paths. With each message, the same decoupled path component is sent from a relay root. For example, an attacker observes that a node u receives $[\mathsf{Path}_i, \mathsf{Payload}_i]$ and sends out $[\mathsf{Path}_{i+1}, \mathsf{Payload}_{i+1}]$ to u_2. Later it observes u receiving Path_i with a different payload, and sending Path_{i+1} with other another payload to u_2. The attacker can then recognize all messages with path component Path_{i+1} as parts of a session involving u and u_2. We simulate the robustness of Cashmere in unlinkability and source anonymity against an attacker observing increasing amounts of network traffic. There are two attacker models: *(i)* the attacker analyzes a fixed fraction of all network traffic, e.g. 0%, 90%, 100%, etc.; or *(ii)* the attacker analyzes a fraction, f_t, of traffic proportional to the fraction of malicious nodes (f) in the network. For example, 10% of malicious nodes can analyze 10% of all traffic. The second is a more realistic model.

We simulate unlinkability and source anonymity for anonymous channels (since it is weaker than anonymous messages), and plot the results in Figures 7 and 8, using parameters $L = 6$. We see that Cashmere is vulnerable to traffic analysis if the attacker observes a significant portion ($> 90\%$) of all network traffic. But Cashmere can still provide high levels of anonymity in the more realistic proportional traffic analysis model.

Cashmere can completely disable traffic analysis attacks with a small modification. Each node in the underlying structured overlay can exchange symmetric keys with peers in its routing table. This sets up secure channels between all node pairs and encrypts all messages using a symmetric cipher. Thus source anonymity and unlinkability are protected against the strongest attacker who can monitor all network traffic. The key-exchange cost is done once per lifetime of a node, in contrast to previous approaches that require per-session key exchanges [11]. Additionally, the small ($O(\log N)$) number of neighbors for each node limits number of key exchanges, whereas approaches like Onion Routing require $O(N^2)$ keys. Finally, the secure channel is established lazily when the first message is routed through that link.

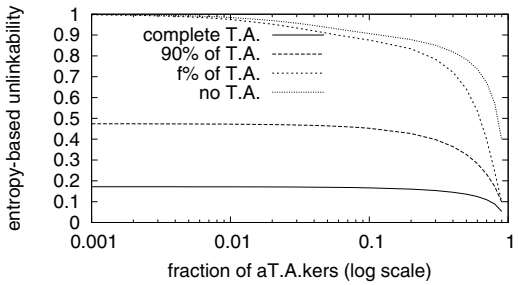


Figure 7: Unlinkability in anonymous channels under different types of traffic analysis (T.A.) with $L = 6$.

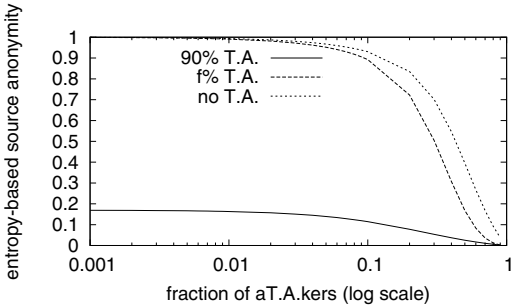


Figure 8: Source anonymity in anonymous channels under different types of traffic analysis (T.A.) with $L = 6$.

5.2 Resilience and Fault-tolerance

Previous anonymous systems use single nodes as relays. Nodes joining and failing in the system can lead to forwarding paths failing. Here we examine the resilience of Cashmere to node churn and intermittent link failures.

We refer to the time between a forwarding relay path is formed and its failure as the *relay path duration*. When a path fails, the sender needs to detect the failure via end-to-end timeouts and establish a new path. If relay path durations are too short, path construction time will dominate. Nodes will constantly be rebuilding failed paths and unable to deliver a message to a destination. Frequent path reconstruction also makes the layer more vulnerable to the degradation attack [32] discussed in Section 2.

In contrast, in Cashmere a relay is usable as long as at least one single node in the network has the relay group's groupID as a prefix. Changes in the membership of the relay group due to node joining and failing are transparent. We first compare the path duration and resilience of Cashmere to previous works.

5.2.1 Churn-resilience

Measurements on real systems have shown that peer-to-peer networks exhibit high node churn [27, 13]. Since most anonymous routing layers are implemented on overlay networks, they must be resilient to high node churn in order to be useful.

Previous studies [25, 13, 27] use *session time* as a metric of churn-rate. We approximate this using an exponential distribution with parameter μ. This churn model is consistent with those used in previous studies of the effect of churn in peer-to-peer systems [15, 25]. Our network model is as follows:

- The network is a finite set (Ω) of nodes, $N = |\Omega|$. The network size is stable, that is, node joins and failures are equal.
- *Session time* is exponentially distributed with parameter μ, meaning node failure is a Poisson process with rate $N\mu$. The mean session time is $\frac{1}{\mu}$ and the median session time is $\frac{\ln 2}{\mu}$.
- *Node arrivals* is a Poisson process with rate λ, where $\lambda = N\mu$.

Previous measurements [27] of file sharing systems suggest median session times of $\frac{\ln 2}{\mu} \approx 60$ min which we used for these experiments.

Figure 9 shows the expected path durations for forwarding paths using relay groups compared to using single nodes as relays. As expected, the use of relay groups increases the expected path duration exponentially, making Cashmere much more resilient to node churn.

5.2.2 Tolerance to Intermittent Failures

We now simulate Cashmere's tolerance to short-term intermittent failures. We model the mean time between failure (MTBF) as $\frac{1}{\lambda_1}$ and mean time to repair (MTTR) as $\frac{1}{\mu_1}$. We assume the failure is a Poisson process with failure event rate λ_1 and time to repair is exponential distributed with parameter μ_1. We assume MTBF $\frac{1}{\lambda_1} = 200$min, and MTTR $\frac{1}{\mu_1} = 5$min.

Figure 10 shows that Cashmere completely masks all intermittent network failures: the expected path duration is more than 10^6 minutes (about 40 days) when we set $\rho = 4$. This is an improvement of several orders of magnitude over previous node-based approaches.

5.2.3 Simulation on Kazaa Measurements

We examine how Cashmere's good path duration properties translate into stability for a real application. We simulate the fetching of objects in a file-sharing application, and examine the number of path repairs required during the object fetches. We model node churn and intermittent failures using parameters from the previous two sections. The distribution of object download times is long-tailed and generated using measurements from the Kazaa network. The Kazaa data [13] has distributions of down-

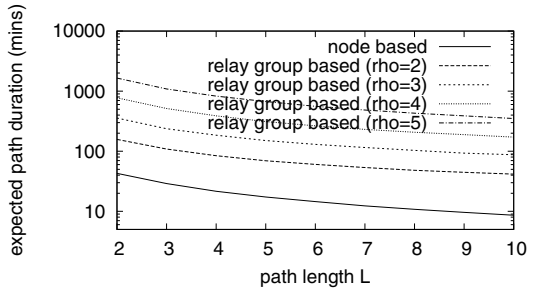


Figure 9: Comparing expected durations of node-based relays and "relay group"-based paths. (Note: ρ is shown as "rho", y-axis is log-scaled.)

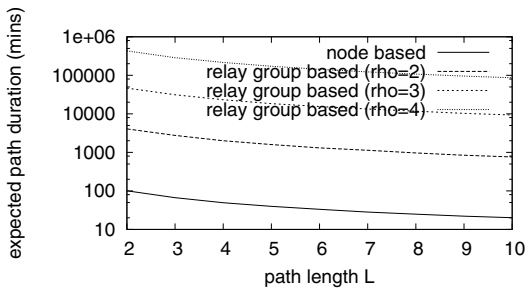


Figure 10: Comparing expected durations of node-based paths and "relay group"-based paths with intermittent failures. (Note: ρ is shown as "rho"; y-axis is log-scaled.)

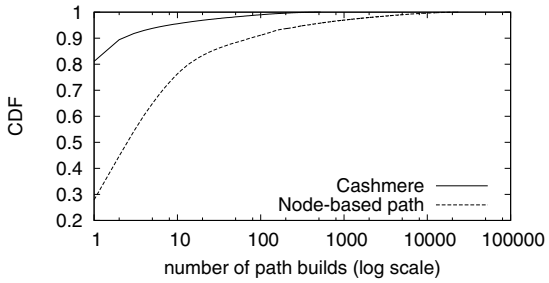


Figure 11: CDF distribution of number of path builds using all downloads in the Kazaa trace, comparing Cashmere (L=6, ρ=5) and node-based relays.

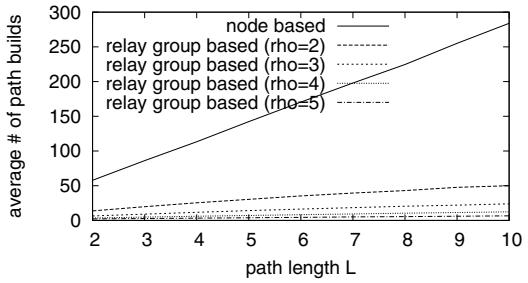


Figure 12: Average number of path builds for small object (10M) downloads using Kazaa data, comparing Cashmere and node-based relays.

load times for small objects (10MB) and large objects (100MB).

We simulate $100,000$ object download sessions on top of both node-based relays and Cashmere's group-based relays. Both systems use relay paths of length $L = 6$, and Cashmere uses average relay group size $\rho = 5$. Using object download times from the Kazaa data, Figure 11 shows the distribution of expected frequencies that each download needs to construct the relay path. It shows that 81% of these small object download sessions using Cashmere would not require any path rebuilds (*i.e.* number of path builds is 1) and no sessions require more than about 500 rebuilds. This compares to 28% using node-based relays, and 10% of all sessions requiring between 100 and 25000 path rebuilds. The maximum number of path builds is very large (*i.e.* 500 and 25000) because Kazaa object download times are long-tail distributed where some objects take extremely long time to download.

The average number of path builds under different parameters (L, ρ) for small object downloads are shown in Figure 12. Clearly, increasing relay group size increases path duration significantly, and Cashmere provides more than an order of magnitude improvement over node-based approaches. Measurements for large file downloads are nearly identical and omitted for brevity.

5.3 Cost and Performance Comparison

In this section we analyze the relative costs in operating Cashmere compared to previous node-based relay approaches. We observe that the operating costs of node-based relay path systems include:

1. Communication costs to maintain knowledge of candidate relays nodes.
2. Bandwidth cost in forwarding messages.
3. Computational costs to construct the relay path at the source and to decrypt messages at intermediate relay nodes.

We first examine communication costs in network maintenance and relay discovery. In node-based relay approaches, nodes are expected to actively maintain information about the other nodes in the network, with a total cost of $O(N^2)$. In contrast, Cashmere decouples maintenance and relay discovery, and relay discovery requires no communication. Nodes estimate the number of nodes in the network by examining their local routing tables, and choose an appropriate prefix length to establish relay groups of average size ρ. Nodes then choose

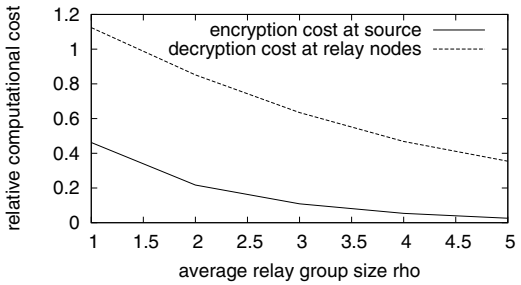


Figure 13: The relative aggregate computational cost over time compared to node-based approaches in a dynamic network. (Note: ρ is shown as "rho" in the figure.)

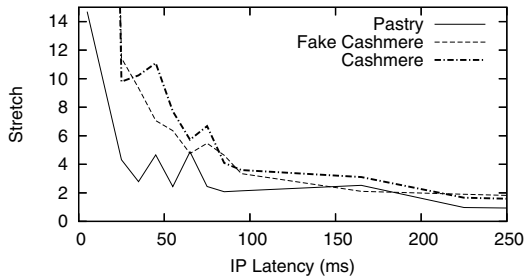


Figure 14: Stretches: Cashmere latency, fake Cashmere latency and Pastry latency vs. IP latency.

random prefixes of the desired length as relay groupIDs. Cashmere relies on the underlying structured overlay, and hence has a total cost of $O(N \log N)$.

However, Cashmere incurs a higher bandwidth cost to gain resilience. Total number of messages sent is $O(\rho L)$ while node-based approaches requires $O(L)$. The extra messages are required to perform the per-relay group broadcast of the payload, and do not adversely impact end-to-end latency or throughput at the overlay layer. This broadcast traffic does contribute to a node's cover traffic that it has to generate.

We now examine computational cost. High per-message computation is often seen as a key obstacle to the wide-spread deployment of Chaum-Mixes based systems. Given a path of length L, a Chaum-Mixes source node performs L asymmetric encryption operations on every message. In addition, each node on the path performs one asymmetric decryption per message that it forwards. The high cost of asymmetric cryptographic operations limits the message send rate at the source and the message forwarding rate at intermediate nodes.

Optimizations have been proposed to reduce computation for session-based communication on Chaum-Mixes by using symmetric key encryption for payload messages and amortizing asymmetric crypto operations across an entire session. Both Tarzan [11] and our solution fall into this category.

Assume the cost of asymmetric encryption and decryption are C_e and C_d respectively. For each relay group path, Cashmere incurs computational cost that includes encryption cost of $L \cdot C_e$ at the source, decryption cost of $2C_d$ at relay group root, decryption cost of C_d at each relay group member, and additional operations to refresh caches after relay group root failures. However, these cost are amortized over a much longer path duration than node-based systems and dwarfed by the cost of rebuilding paths in node-based systems.

Based on previous results of expected durations, Figure 13 plots the cost of our "relay group"-based approach relative to that of node-based session solutions on a realistic network. As ρ increases, the path duration increases and the per-session cost drops. For $\rho = 4$, the encryption cost at the source in Cashmere is roughly 5.37% of the cost at source nodes in node-based solutions. The aggregate decryption cost at relay group members in Cashmere is 46.83% of the cost at intermediate nodes in node-based solutions. The reduction in encryption computation is from amortizing the one time path setup costs across the long path durations of Cashmere. The reduction in decryption costs is from per-node caching of the path component and whether a node is the destination, and reducing the number of asymmetric crypto operations to just one per session for nodes who are not the destination.

5.4 Implementation Measurements

We ran experiments to determine the latency, throughput and computational overheads of Cashmere.

We deployed and evenly distributed 128 Cashmere nodes on 32 machines from PlanetLab that are geographically distributed all over the United States. We define groupIDs to be 5-bit prefixes, so relay groups have average size of 4 nodes. We measure latency in:

- *Cashmere*: End to end latency of Cashmere routing across 4 relay groups;
- *Fake Cashmere*: End to end latency of Cashmere routing across 4 relay groups, removing cryptographic computation;
- *Pastry*: The latency of routing via Pastry directly from source to destination;
- *IP*: Direct IP latency.

Message payloads are 24 bytes long. The latency is measured using round trip time (RTT), by sending messages from one node to all other nodes with each repeated 10 times.

We show the average latency in Cashmere, Fake Cashmere, Pastry vs. direct IP latency in Figure 14. The "stretch" is computed as each sample of Cashmere/Fake

Cashmere/Pastry latency divided by average IP latency for the same destination. To plot the graph, we put all stretch samples into bins of 10ms intervals of average IP latency. Figure 14 shows that the stretches decrease while the IP latency between source and destination increases. For a pair of end nodes that are very close to each other (i.e. < 50ms), Cashmere stretches are about two times of Pastry. The extra delays introduced by the Cashmere layer is significant compared to small IP latency values. Most samples of IP latency are from 50ms to 250ms. In this range, Cashmere stretches are between 1.9 to 5.5, which is quite close to Pastry (2.1 to 4.8). This means Cashmere layer introduces a relatively small delay on the overlay. Comparing stretches between Cashmere and fake Cashmere shows that delay caused by cryptographic computation in Cashmere is negligible. This is attributed to no per message asymmetric encryption/decryption in Cashmere. We also measured that the average number of IP messages per Cashmere message is 19.54 and the average number of IP messages per Pastry message is 1.54. The larger number of IP message comes from the relay and broadcast messages in Cashmere.

To measure computation cost, we utilize FreePastry's network emulation capabilities. We created 64 virtual FreePastry nodes inside the same Java virtual machine on a 2.4Ghz Pentium IV PC. The virtual nodes are connected together using local loopback (called "direct" network in FreePastry) network transport. There is no CPU contention between the nodes because the emulation is event-driven and at most one virtual node is running at a time. Cashmere is set up similarly as above. We obtain highly accurate time measurements by calling the `RDTSC` instruction supported by the Pentium architecture via Java Native Interface (JNI).

In the first experiment, we approximate throughput of relay group roots by measuring per-message latency across 1000 random source-destination pairs. For each source and destination pair, we send a single message to set up the path and allow relay group roots to set up their caches, then measure the latency taken to process a second payload message. We then approximate the throughput as $\frac{1}{latency}$. Table 2 shows the results for forwarding throughput of relay group roots for different message sizes.

In the second experiment we measure the computational overheads for the source, the relay group root nodes, the non-root relay group nodes and the destination, for both the first and subsequent messages. 1000 empty messages are sent from random source to destination with and without the routes already set up. Table 3 summarizes the results, showing the average CPU time incurred per node role with the standard deviation in brackets. The first message invokes RSA on each hop and therefore is relatively expensive. The subsequent messages to the same destination, using the same forwarding path, utilize cached routing information on each node. Therefore they only invoke Blowfish which is less expensive.

Msg Size (B)	Msg/second	Throughput (Mb/s)
128	1370	1.337
1024	1160	9.063
4096	855	26.72
16384	386	48.25

Table 2: Message forwarding rate and effective throughput for different message sizes of relay group root nodes.

	First Msg	Subsequent Msg
Source	8.21 (5.3)	0.73 (0.39)
Relay Group Root	27.5 (11.8)	0.22 (0.10)
Non-root Group Member	4.73 (347)	0.001 (0.05)
Destination	7.19 (1.87)	0.18 (0.03)

Table 3: CPU time (ms) spent by each class of node routing an empty message using Cashmere. Standard deviation shown in parentheses.

We also evaluated the space overhead during the experiment. At the source nodes the overhead for each message is 456 bytes for the path element and any necessary padding bytes to round the payload to RSA block sizes (64 bytes).

6 Conclusion

We present Cashmere, a resilient anonymous routing infrastructure that leverages the flexible anycast routing inherent in structured overlay networks to significantly improve path durations compared to node-based relay approaches. Cashmere also decouples the encrypted path component of each session from the payload, and uses symmetric session keys to encrypt message payloads. Anonymous source nodes in Cashmere can choose their own per-session parameters to tradeoff between anonymity, resilience and computation overhead.

We compare Cashmere to previous node-based Chaum-Mixes approaches through analysis and simulation. We find that Cashmere provides similar anonymity properties while providing one to two orders of magnitude improvement in path durations under both node churn and intermittent failures. This translates into significantly lower path reconstructions across an anonymous application session. Performance optimizations in Cashmere avoid asymmetric crypto operations, resulting in lower per-session computation costs compared to other session-based Chaum-Mixes approaches. Finally, we provide measurements of a real Cashmere deployment and show that it provides reasonable throughput

while incurring a small latency overhead over structure overlay routing.

Ongoing work on Cashmere includes issues related to key management and key revocation in particular. We are also interested in better understanding the impact of network dynamics on key discovery. A straight-forward yet very useful extension to Cashmere is to support anonymous object location in DOLR [6, 34] overlays like Pastry and Tapestry. Finally, we are working on a stable wide-area deployment on PlanetLab and a software package for public release.

Acknowledgements

We would like to thank the anonymous reviewers for their comments and Vern Paxson our shepherd. We also thank Stefan Saroiu and Krishna Gummadi for providing us with data on filesharing session statistics.

Notes

[1]Cashmere only requires each node generates a small amount of traffic. When the real traffic is not sufficient, nodes send out dummy messages as cover traffic.

[2]Nodes in Ω_i are equal, each with probability $\frac{\Pr(\Omega_i)}{|\Omega_i|}$ to be the source (or destination).

[3]This is a worst case assumption. In reality the attacker can only estimate this by monitoring certain network latencies and system overheads. For example, the more relay groups are used, the more computation a source will perform.

References

[1] The anonymizer. Anonymizer.com.

[2] CASTRO, M., DRUSCHEL, P., GANESH, A., ROWSTRON, A., AND WALLACH, D. S. Security for structured peer-to-peer overlay networks. In *Proc. of OSDI* (Dec 2002).

[3] CHAUM, D. L. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM 24*, 2 (1981).

[4] COSTA, M., CASTRO, M., AND ROWSTRON, A. Performance and dependability of structured peer-to-peer overlays. In *Proc. of DSN* (Jun 2004).

[5] CRYPTIX TEAM. Cryptix. `http://www.cryptix.org/`.

[6] DABEK, F., ZHAO, B., DRUSCHEL, P., KUBIATOWICZ, J., AND STOICA, I. Towards a common API for structured P2P overlays. In *Proc. of IPTPS* (Feb 2003).

[7] DANEZIS, G. Mix-networks with restricted routes. In *Proc. of PET* (Mar 2003).

[8] DIAZ, C., SEYS, S., CLAESSENS, J., AND PRENEEL, B. Towards measuring anonymity. In *Proc. of PET* (Apr 2002).

[9] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Tor: The second-generation onion router. In *Proc. of the USENIX Security Symposium* (Aug 2004).

[10] DINGLEDINE, R., AND SYVERSON, P. Reliable MIX cascade networks through reputation. In *Proc. of FC* (Mar 2002).

[11] FREEDMAN, M. J., AND MORRIS, R. Tarzan: A peer-to-peer anonymizing network layer. In *Proc. of CCS* (Nov 2002).

[12] Freepastry. `http://freepastry.rice.edu/`.

[13] GUMMADI, K. P., DUNN, R. J., SAROIU, S., GRIBBLE, S. D., LEVY, H. M., AND ZAHORJAN, J. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proc. of SOSP* (Oct 2003).

[14] HILDRUM, K., AND KUBIATOWICZ, J. Asymptotically efficient approaches to fault-tolerance in peer-to-peer networks. In *Proc. of DISC* (Oct 2003).

[15] LIBEN-NOWELL, D., BALAKRISHNAN, H., AND KARGER, D. Analysis of the evolution of peer-to-peer systems. In *Proc. of PODC* (Jul 2002).

[16] MISLOVE, A., OBEROI, G., POST, A., REIS, C., DRUSCHEL, P., AND WALLACH, D. S. Ap3: Cooperative, decentralized anonymous communication. In *Proc. of SIGOPS European Workshop* (Sep 2004).

[17] MÖLLER, B. Provably secure public-key encryption for length-preserving chaumian mixes. In *Proc. of CT-RSA* (Apr 2003).

[18] MÖLLER, U., COTTRELL, L., PALFRADER, P., AND SASSAMAN, L. Mixmaster Protocol — Version 2. Draft, Jul 2003.

[19] NGAN, T., WALLACH, D., AND DRUSCHEL, P. Enforcing fair sharing of peer-to-peer resources. In *Proc. of IPTPS* (Feb 2003).

[20] PFITZMANN, A., AND KÖHNTOPP, M. Anonymity, unobservability and pseudonymity. In *Proc. of the Intl. Workshop on the Design Issues in Anonymity and Observability* (Jul 2000).

[21] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SCHENKER, S. A scalable content-addressable network. In *Proc. of SIGCOMM* (Aug 2001).

[22] REED, M. G., SYVERSON, P. F., AND GOLDSCHLAG, D. M. Anonymous connections and onion routing. *IEEE JSAC 16*, 4 (May 1998).

[23] REITER, M., AND RUBIN, A. Crowds: Anonymity for web transactions. *ACM Trans. on Inf. and Syst. Secur. 1*, 1 (Jun 1998).

[24] RENNHARD, M., AND PLATTNER, B. Introducing MorphMix: Peer-to-Peer based Anonymous Internet Usage with Collusion Detection. In *Proc. of WPES* (Nov 2002).

[25] RHEA, S., GEELS, D., ROSCOE, T., AND KUBIATOWICZ, J. Handling churn in a DHT. In *Proc. of USENIX* (Jun 2004).

[26] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of Middleware* (Nov 2001).

[27] SAROIU, S., GUMMADI, P. K., AND GRIBBLE, S. A measurement study of peer-to-peer file sharing systems. In *Proc. of MMCN* (Jan 2002).

[28] SERJANTOV, A., AND DANEZIS, G. Towards an information theoretic metric for anonymity. In *Proc. of PET* (Apr 2002).

[29] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of SIGCOMM* (Aug 2001).

[30] SYVERSON, P., TSUDIK, G., REED, M., AND LANDWEHR, C. Towards an analysis of onion routing security. In *Proc. of PET* (Jul 2001).

[31] SYVERSON, P. F., GOLDSCHLAG, D. M., AND REED, M. G. Anonymous connections and onion routing. In *IEEE Symposium on Security and Privacy* (May 1997).

[32] WRIGHT, M., ADLER, M., LEVINE, B. N., AND SHIELDS, C. An analysis of the degradation of anonymous protocols. In *Proc. of NDSS* (Feb 2002).

[33] WRIGHT, M. K., ADLER, M., LEVINE, B. N., AND SHIELDS, C. The predecessor attack: An analysis of a threat to anonymous communications systems. *ACM Trans. Inf. Syst. Secur. 7*, 4 (2004).

[34] ZHAO, B. Y., HUANG, L., RHEA, S. C., STRIBLING, J., JOSEPH, A. D., AND KUBIATOWICZ, J. D. Tapestry: A global-scale overlay for rapid service deployment. *IEEE J-SAC 22*, 1 (Jan 2004).

[35] ZHAO, B. Y., HUANG, L., STRIBLING, J., JOSEPH, A. D., AND KUBIATOWICZ, J. D. Exploiting routing redundancy via structured peer-to-peer overlays. In *Proc. of ICNP* (Nov 2003).

Decentralized, Adaptive Resource Allocation for Sensor Networks

Geoffrey Mainland, David C. Parkes, and Matt Welsh
Division of Engineering and Applied Sciences
Harvard University
{mainland,parkes,mdw}@eecs.harvard.edu

Abstract

This paper addresses the problem of resource allocation in sensor networks. We are concerned with how to allocate limited energy, radio bandwidth, and other resources to maximize the value of each node's contribution to the network. Sensor networks present a novel resource allocation challenge: given extremely limited resources, varying node capabilities, and changing network conditions, how can one achieve efficient global behavior? Currently, this is accomplished by carefully tuning the behavior of the low-level sensor program to accomplish some global task, such as distributed event detection or in-network data aggregation. This manual tuning is difficult, error-prone, and typically does not consider network dynamics such as energy depletion caused by bursty communication patterns.

We present *Self-Organizing Resource Allocation (SORA)*, a new approach for achieving efficient resource allocation in sensor networks. Rather than manually tuning sensor resource usage, SORA defines a virtual market in which nodes sell *goods* (such as sensor readings or data aggregates) in response to *prices* that are established by the programmer. Nodes take actions to maximize their profit, subject to energy budget constraints. Nodes individually adapt their operation over time in response to feedback from payments, using reinforcement learning. The behavior of the network is determined by the price for each good, rather than by directly specifying local node programs.

SORA provides a useful set of primitives for controlling the aggregate behavior of sensor networks despite variance of individual nodes. We present the SORA paradigm and a sensor network vehicle tracking application based on this design, as well as an extensive evaluation demonstrating that SORA realizes an efficient allocation of network resources that adapts to changing network conditions.

1 Introduction

Sensor networks, consisting of many low-power, low-capability devices that integrate sensing, computation, and wireless communication, pose a number of novel systems problems. They raise new challenges for efficient communication protocols [13, 44], distributed algorithm design [11, 24], and energy management [1, 5]. While a number of techniques have been proposed to address these challenges, the general problem of resource allocation in sensor networks under highly volatile network conditions and limited energy budgets remains largely unaddressed. Current programming models require that global behavior be specified in terms of the low-level actions of individual nodes. Given varying node locations, capabilities, energy budgets, and time-varying network conditions, this approach makes it difficult to tune network-wide resource usage. We argue that new techniques are required to bridge the gap from high-level goals to low-level implementation.

In this paper, we present a novel approach to adaptive resource allocation in sensor networks, called *Self-Organizing Resource Allocation (SORA)*. In this approach, individual sensor nodes are modeled as self-interested agents that attempt to maximize their "profit" for performing local actions in response to globally-advertised price information. Sensor nodes run a very simple cost-evaluation function, and the appropriate behavior is induced by advertising prices that drives nodes to react. Nodes adapt their behavior by learning their utility for each potential action through payment feedback. In this way, nodes dynamically react to changing network conditions, energy budgets, and external stimuli. Prices can be set to meet systemwide goals of lifetime, data fidelity, or latency based on the needs of the system designer.

Consider environmental monitoring [9, 28] and distributed vehicle tracking [23, 45], which are two oft-cited applications for sensor networks. Both applications require nodes to collect local sensor data and relay it to a central base station, typically using a multihop routing scheme. To reduce bandwidth requirements, nodes may need to aggregate their local sensor data with that of other nodes. In-network detection of distributed phenomena, such as gradients and isobars, may require more sophisticated cross-node coordination [15, 41, 47].

Two general challenges emerge in implementing these applications. First, nodes must individually determine the ideal rate for sampling, aggregating, and sending data to operate within some fixed energy budget. This rate affects overall lifetime and the accuracy of the results produced by the network. Each node's ideal schedule is based on its physical location, position in the routing topology, and changes in environmental stimuli. Many current applications use a fixed schedule for node actions, which is suboptimal when nodes are differentiated in this way. Second, the system may wish to tune these schedules in response to changes in the environment, such as the target vehicle's location and velocity, to meet goals of data rate and latency. More complex adaptations might involve selectively activating nodes that are expected to be near some

phenomenon of interest. Currently, programmers have to implement these approaches by hand and have few tools to help determine the ideal operating regime of each node.

Rather than defining a fixed node schedule, SORA causes nodes to individually tune their rate of operation using techniques from reinforcement learning [37]. Nodes receive virtual payments for taking "useful" actions that contribute to the overall network goal, such as listening for incoming radio messages or taking sensor readings. Each node learns which actions are profitable based on feedback from receiving payments for past actions. Network retasking is accomplished by adjusting prices, rather than pushing new code to sensor nodes. Network lifetime is controlled by constraining nodes to take actions that meet a local energy budget.

In this paper, we focus on a specific challenge application, vehicle tracking, which provides a rich space of problems in terms of managing latency, accuracy, communication overhead, and task decomposition. The SORA model is not specifically tailored to tracking, however, and can be readily adopted for other problem domains. We present a thorough evaluation of the SORA approach using a realistic sensor network simulator. Our results demonstrate that, using SORA, resource allocation within the sensor network can be controlled simply by advertising prices, and that nodes self-organize to take the set of actions that make the greatest contribution to the global task under a limited energy budget. This paper expands on our previous workshop paper [18] on SORA, presenting a thorough evaluation of the approach.

We show that SORA achieves more efficient allocation than static node scheduling (the most commonly-used approach currently in use), and outperforms a dynamic scheduling approach that accounts for changes in energy availability. In addition, SORA makes it straightforward to differentiate node activity by assigning price vectors that influence nodes to select certain actions over others.

The rest of this paper is organized as follows. In Section 2 we present the background for the SORA approach, specific goals, and related work. Section 3 presents the Self-Organizing Resource Allocation model in detail, and Section 4 describes the use of SORA in our vehicle tracking application. Section 5 presents our implementation of SORA in a realistic sensor network simulator, as well as evaluation in terms of network behavior and node specialization as prices, energy budgets, and other parameters are tuned. Finally, Section 6 describes future work and concludes.

2 Motivation and Background

Sensor networks consist of potentially many nodes with very limited computation, sensing, and communication capabilities. A typical device is the UC Berkeley Mica2 node, which consists of a 7.3 MHz ATmega128L processor, 128KB of code memory, 4KB of data memory, and a Chipcon CC1000 radio capable of 38.4 Kbps and an outdoor transmission range of approximately 300m. The node measures 5.7cm $\times$ 3.1cm $\times$ 1.8cm and is typically powered by 2 AA batteries with an expected lifetime of days to months, depending on application duty cycle. The limited memory and computational resources of this platform make an interesting design point, as software layers must be tailored for this restrictive environment. The Mica2 node uses a lean, component-oriented operating system, called TinyOS [16], and an unreliable message-passing communication model based on Active Messages [38].

To begin, we outline the distributed resource allocation problem that arises in the sensor network domain. We highlight several prior approaches to this problem and make the case for market-based techniques as an attractive solution.

2.1 Resource allocation in sensor networks

Sensor networks have been proposed for a wide range of novel applications. Examples include instrumenting buildings, bridges, and other structures to measure response to seismic events [8, 20], monitoring environmental conditions and wildlife habitats [9, 28], tracking of vehicles along a road or in an open area [43], and real-time monitoring of patient vital signs for emergency and disaster response [25, 33].

One of the core challenges of sensor application design is balancing the resource usage of individual nodes with the global behavior desired of the network. In general, the sequence of actions taken by a node affects local energy consumption, radio bandwidth availability, and overall quality of the results. However, tuning the resource usage of individual sensor nodes by hand is difficult and error-prone. Although TinyOS [16] and other systems provide interfaces for powering down individual hardware devices such as the radio and CPU, using these interfaces in a coordinated fashion across the network requires careful planning. For example, if a node is sleeping, it cannot receive or route radio messages.

The typical approach to scheduling sensor operations is to calculate a static schedule for all nodes in the network. For example, query-based systems such as TinyDB [26] and Cougar [46] allow the user to specify a *query epoch* that drives periodic sampling, aggregation, and data transmission. Other programming models, such as directed diffusion [12, 17], abstract regions [41], or Hoods [43], either assume periodic data collection or leave scheduling to higher-level code. However, an application that uses a fixed schedule for every node will exhibit very different energy consumption rates across the network. For example, nodes responsible for routing messages will consume more energy listening for and sending radio messages. Likewise, nodes on the network periphery may not need to route radio messages at all.

Another solution is to compute, offline, the optimal schedule for each node based on a model of radio con-

nectivity, node location, and physical stimuli that induce network activity. For example, Adlakha *et al.* [1] describe a design-time recipe for tuning aspects of sensor networks to achieve given accuracy, latency, or lifetime goals. However, this approach assumes a statically-configured network where resource requirements are known in advance, rather than allowing the network behavior to be tuned at runtime (say, in response to increased activity).

Other systems have attempted to address the node scheduling problem for specific applications or communication patterns. For example, Liu *et al.* [24] describe an approach to tracking a moving half-plane shadow through a sensor network that can be used to selectively activate nodes along the frontier of the shadow. STEM [34] is a protocol that dynamically wakes nodes along a routing path to trade energy consumption for latency. LEACH [13] is a cluster-based routing scheme that rotates the local clusterhead to distribute energy load across multiple nodes. These techniques point to more general approaches to adapting the behavior of sensor networks to maximize lifetime.

Providing application control over resource usage is often desirable when designing high-level programming abstractions for sensor networks. Abstract regions [41] focuses on the ability to tune the communication layer to trade off energy for accuracy. Likewise, TinyDB provides a *lifetime* keyword that scales the query sampling and transmission period of individual nodes to meet a user-supplied lifetime target [27]. Both of these approaches provide a means for nodes to "self-tune" their behavior to meet specific systemwide resource and accuracy targets. However, the general problem of adaptive resource allocation in sensor networks has not been adequately addressed.

2.2 AI-based approaches to resource allocation

The SORA approach draws on the areas of reinforcement learning and economic theory to yield new techniques for decentralized optimization in sensor networks. In *reinforcement learning* [37], an agent attempts to maximize its "reward" for taking a series of actions. Whether or not a node receives a reward is defined by the *success* of the action; for example, whether a radio message is received while the node is listening for incoming messages. The agent's goal is to maximize its reward, subject to constraints on resource usage, such as energy.

The reward for each successful action can be modeled as a *price* in a virtual market. By applying ideas from economic theory, SORA attempts to achieve efficient resource allocation in a decentralized fashion. Economics has been used as an inspiration for solving resource-management problems in many computational systems, such as network bandwidth allocation [35], distributed database query optimization [36], and allocating resources in distributed systems such as clusters, Grids, and peer-to-peer networks [3, 4, 7, 10, 39].

Much of this prior work has been concerned with resource arbitration across multiple *self-interested* users, which may attempt to cheat or otherwise hoard resources in the system for their own advantage. In the sensor network context, however, we assume that nodes are well-behaved and *program* them to behave as the classic economic actors of microeconomic theory. *Thus, we use markets as a programming paradigm*, not because we are concerned with self-interested behavior of sensor nodes. We need not model complex game-theoretic behavior, but can instead focus on nodes that (by design) are classic price-taking economic agents.

SORA is inspired by Wellman's seminal work on *market-oriented programming* [30, 40], which uses market equilibrium to solve statically-defined distributed optimization problems. We believe that SORA is the first serious attempt to use market-oriented methods to provide complete runtime control for a real distributed system. This systems focus leads us to consider continuous, real-time resource allocation, while Wellman's work was concerned with solving a static allocation problem. Other recent work has applied economic ideas to specific sensor network problems. For instance, market-inspired methods have been suggested for the problems of *ad hoc* routing [2] and information-directed query processing [47]. Our goal in SORA is not to provide a point solution but to address the general issue of adaptive resource allocation.

3 Self-Organizing Resource Allocation

In Self-Organizing Resource Allocation (SORA), sensor nodes are programmed to maximize their "profit" by taking actions subject to energy constraints. Actions that contribute to the network's overall goal, such as taking useful sensor readings or forwarding radio messages, result in a payment to the node taking the action. By setting the price for each action, the network's global behavior can be tuned by the system designer. Nodes continuously learn a model for which actions are profitable, allowing them to adapt to changing conditions.

3.1 Goals

The essential problem that SORA addresses is that of determining the set of local actions to be taken by each sensor node to meet some global goals of lifetime, latency, and accuracy for the data produced by the network as a whole. Each node can take a set of local actions (such as data sampling, aggregation, or routing), each with varying energy costs and differing contributions to the global task of the network. Through *self-scheduling*, nodes independently determine their ideal behavior (or *schedule*) subject to constraints on local energy consumption. Self-scheduling in SORA meets three key goals:

Differentiation: Nodes in a sensor network are heterogeneous in terms of their position in the network topology, resource availability, and proximity to phenomena of interest. Through self-scheduling, nodes *differentiate* their behavior based on this variance. For example, nodes closer

to phenomena of interest should acquire and transmit more sensor readings than those nodes that are further away.

Adaptivity: Differentiation in nodal behavior should also vary with time, as external stimuli move, nodes join and leave the network, energy reserves are depleted, and network connectivity shifts. Such adaptation permits a more efficient use of network resources. For example, nodes will consume energy only when it is worthwhile to do so based on current network conditions, rather than as dictated by an *a priori* schedule.

Control: Finally, a system designer should have the ability to express systemwide goals and effect *control* over the behavior of the network despite uncertainty in the exact state, energy reserves, and physical location of sensor nodes. For example, if the data rate being generated by the network is insufficient for the application's needs, nodes should be instructed to perform sampling and routing actions more frequently. This goal differs from internal adaptation by nodes, since it requires external observation and manipulation.

3.2 SORA overview

In the SORA model, each sensor node acts as an *agent* that attempts to maximize its profit for taking a series of *actions*. Each action consumes some amount of *energy* and produces one or more *goods* that have an associated *price*. Nodes receive payments by producing goods that contribute value to the network's overall operation. For example, a node may be paid for transmitting a sensor reading that indicates the proximity of a target vehicle, but not be paid if the vehicle is nowhere nearby. Reacting to this payment feedback is the essential means of adaptivity in SORA. Prices are determined by the *client* of the sensor network, which can be thought of as an external agent that receives data produced by the network and sets prices to induce network behavior.

The local program executed by each node is simple and avoids high communication overhead in order to operate efficiently. In the SORA approach, nodes operate using primarily *local* information about their state, such as energy availability. The only *global* information shared by nodes is the current set of prices, which are defined by the sensor network client. To minimize overhead, prices should be updated infrequently (for example, to effect large changes in the system's activity) and can be propagated to nodes through a variety of efficient gossip or controlled-flooding protocols [22].

3.3 Goods and actions

The actions that sensor nodes can take depend on the application, but typically include sampling a sensor, aggregating multiple sensor readings, or broadcasting a radio message. An action may be *unavailable* if the node does not currently have enough energy to perform the action. In addition, production of one good may have dependencies on the availability of others. For example, a node cannot aggregate sensor readings until it has acquired multiple readings.

Taking an action may or may not produce a good of value to the sensor network as a whole. For example, listening for incoming radio messages is only valuable if a node hears a transmission from another node. Likewise, transmitting a sensor reading is only valuable if the reading has useful informational content. We assume that nodes can determine locally whether a given action deserves a payment. This works well for the simple actions considered here, although more complex actions (e.g., computing a function over a series of values) may require external notification for payments.

3.4 Energy budget

A node's *energy budget* constrains the actions that it can take. We assume that nodes are aware of how much energy each action takes, which is straightforward to measure offline. The energy budget can be modeled in a number of different ways. A simple approach is to give each node a fixed budget that it may consume over an arbitrary period of time. In this case, however, nodes may rapidly deplete their energy resources by taking many energy-demanding actions, resorting to less-demanding actions only when reserves get low.

To capture the desire for nodes to consume energy at a regular rate, we opt to use a token bucket model for the budget. Each node has a bucket of energy with a maximum capacity of C Joules, filling at a rate ρ that represents the average desired rate of energy usage (e.g., 1000 J per day). When a node takes an action, the appropriate amount of energy is deducted from the bucket. If a node cannot take any action because its bucket is too low, it must *sleep*, which places the node in the lowest-possible energy state.

The capacity C represents the total amount of energy that a node can consume in one "burst." If C is set to the size of the node's battery, the node is able to consume its entire energy reserves at once. By limiting C, one can bound the total amount of energy used by a node over a short time interval.

3.5 Agent operation

Given a set of actions, goods produced by those actions, prices for each good, and energy cost for each action, each agent operates as follows. A node simply monitors its local state and the global price vector, and periodically selects the action that maximizes its *utility* for each action. Upon taking that action, the node's energy budget is reduced by the appropriate amount, and the node may or may not receive a payment depending on whether its action produced a valuable good. We define the utility function $u(a)$ for an action a to be:

$$u(a) = \begin{cases} \beta_a p_a & \text{if the action is } \textit{available} \\ 0 & \text{otherwise} \end{cases}$$

where p_a is the current price for action a, and β_a is the *estimated probability of payment* for that action, which is learned by nodes as described below. An action may be *unavailable* if either the current energy budget is too low to take the action, or other dependencies have not been met (such as lack of sensor readings to aggregate).

In essence, the utility function represents the expected profit for taking a given action. The parameter β_a is continuously estimated by nodes over time in response to the success of taking each action. This is a form of reinforcement learning [37]. After taking an action a, the new value β'_a is calculated based on whether the action received a payment:

$$\beta'_a = \begin{cases} \alpha + (1-\alpha)\beta_a & \text{if } a \text{ receives a payment} \\ (1-\alpha)\beta_a & \text{otherwise} \end{cases}$$

α represents the sensitivity of the EWMA filter (in our experiments, $\alpha = 0.2$). In this way, nodes learn which actions are likely to result in payments, leading to a natural self-organization depending on the node's location in the network or intrinsic capabilities. For example, a node that has the opportunity to route messages for other nodes will be paid for listening for incoming radio messages; nodes on the edges of the network will learn that this action is rarely (if ever) profitable.

The expected profit for an action will vary over time due to price adjustments and changing environmental conditions. Therefore, it is important that nodes periodically "take risks" by choosing actions that have a low payment probability β_a. We employ an ϵ-greedy action selection policy. That is, with probability $1-\epsilon$ (for some small ϵ; we currently use $\epsilon = 0.05$), nodes select the "greedy" action that maximizes the utility $u(a)$. However, with probability ϵ a node will select an (available) action from a uniform distribution. In effect, this ignores the value of β_a and allows a node to explore for new opportunities for profit. Such exploration prevents a node from never electing to take an action because it has not recently been paid to do so [37].

Our current reinforcement learning scheme does not take into consideration other aspects of a node's state, such as the sequence of past actions or the state of neighboring nodes, which may lead to more efficient solutions. However, these techniques involve considerable complexity, which goes against our goals of simplicity and limiting per-node state. We intend to explore alternative learning algorithms as part of future work.

3.6 Price selection and adjustment

In SORA, the global behavior of the network is controlled by the client establishing prices for each good. Prices are propagated to sensor nodes through an efficient global data dissemination algorithm, such as SPIN [14] or Trickle [22]. The client can also adjust prices as the system runs, for example, to affect coarse changes in system activity.

There is a complex relationship between prices and agent behavior. Raising the price for a good will not necessarily induce more nodes to produce that good; the dynamics of maximizing expected profits may temper a node's desire to take a given action despite a high price. Our experiments in Section 5 demonstrate the effect of varying prices. As it turns out, subtle changes to prices do not have much impact on global network behavior. This is because each node's operation is mostly dictated by its adaptation to coarse-grained changes in the local state, such as whether sampling sensors or listening for incoming radio messages is currently profitable. Prices serve to differentiate behavior only when a node has multiple profitable actions to choose between. Even when one action has a much higher price than others, nodes will still take a mixture of actions due to continual exploration of the state space through the ϵ-greedy learning policy.

Action	Energy consumed
sample (single sensor)	$8.41 \times 10^{-5} J$
send (single message)	$2.45 \times 10^{-3} J$
listen	$5.97 \times 10^{-3} J$
sleep	$8.25 \times 10^{-5} J$
aggregate (compute max of array)	$8.41 \times 10^{-5} J$

Figure 1: **Energy consumed for each sensor action.**

The best approach to selecting optimal price settings in SORA is still an open problem. Given the complexities of agent operations and unknown environmental conditions, analytically solving for prices to obtain a desired result is not generally possible. In a stationary system, it is possible to search for optimal prices by slowly adjusting each price and observing its effect on network behavior; this approach is used by the WALRAS system [40].

A better approach is to determine prices empirically based on an observation of the network's behavior at different price points. For example, a system designer can experiment with a testbed deployment or simulation to understand the effect of differing prices on overall behavior. Prices can be readily tuned after deployment, since broadcasting a new price vector to an active network is not expensive. This process could be automated by an external controller that observes the network behavior over time and adjusts prices accordingly.

One approach to setting prices, based on economic principles, is to establish a *competitive equilibrium*, where the supply of goods produced by the network equals the demand for those goods expressed by the client. This model is attractive when there are multiple users programming the sensor network to take different sets of actions on their behalf, since equilibrium prices ensure that network resources are shared in an optimal manner. However, computing equilibrium prices often requires continuous information on the network's supply of goods, which may lag pricing updates. A detailed discussion of this technique is beyond the scope of this paper, but we return to this problem in Section 6.

4 Application Example: Vehicle Tracking

As a concrete example of using SORA to manage resource allocation in a realistic sensor network application, we consider tracking a moving vehicle through a field of sensors. We selected vehicle tracking as a "challenge application" for SORA because it raises a number of interesting problems in terms of detection accuracy and latency, in-network aggregation, energy management, routing, node specialization, and adaptivity [6, 43, 45]. Vehicle tracking can be seen as a special case of the more general data collection problem also found in applications such as environmental and structural monitoring [20, 28].

4.1 Tracking overview

In the tracking application, each sensor is equipped with a magnetometer capable of detecting local changes in magnetic field, which indicates the proximity of the vehicle to the sensor node. One node acts as a fixed base station, which collects readings from the other sensor nodes and computes the approximate location of the vehicle based on the data it receives. The systemwide goal is to track the location of the moving vehicle as accurately as possible while meeting a limited energy budget for each node.

Each sensor node can take the following set of actions: *sample* a local sensor reading, *send* data towards the base station, *listen* for incoming radio messages, *sleep* for some interval, and *aggregate* multiple sensor readings into a single value. Each node maintains a fixed-length LIFO buffer of sensor readings, which may be sampled locally or received as a radio message from another node. Each entry in the buffer consists of a tuple containing a vehicle location estimate weighted by a magnetometer reading. The sample action appends a local reading to the buffer, and the listen action may add an entry if the node receives a message from another node during the listen interval.

Aggregation is used to limit communication bandwidth by combining readings from multiple nodes into a single value representing the "best" sensor reading. The aggregate action replaces the contents of the sample buffer with a single weighted position estimate, ignoring any sample older than a programmer-defined constant (10 sec in our simulations). The *sleep* action represents the lowest-energy state of a node which is entered when energy is unavailable for other actions, or no other action is deemed profitable. Figure 1 summarizes the energy requirements for each action, based on measurements of the Mica2 sensor node.

4.2 Routing

All radio transmissions route messages towards the base station using a multihop routing protocol. Nodes are not assumed to be within a single radio hop of the base. The choice of routing algorithm is not essential; we use a simple greedy geographic routing protocol, similar to GPSR [19] but without any face routing, although other routing algorithms can be used [44, 17]. Messages are forwarded to the neighboring node that is both physically closer to the destination (always the base station, in this case) and is currently executing the *listen* action. This protocol assumes a CTS-RTS MAC layer that allows a node to send a message to any one of its next-hop neighbors that are currently listening. In this way, as long as any closer neighbor is currently listening, the message will be forwarded. This approach meshes well with the stochastic nature of node actions in SORA and does not require explicit coscheduling of senders and receivers.

4.3 Discussion

SORA naturally leads to an efficient allocation of network resources. Individual nodes are constrained to operate within their energy budget, and the schedule for each node may vary over time depending on network conditions and external stimuli. Nodes continuously learn which actions are most profitable and thereby have the most value to the sensor network as a whole. This *emergent* behavior is more effective at allocating limited network resources than traditional schemes based on static schedules.

The SORA approach captures a number of design tradeoffs that are worth further discussion. One advantage of this model is that the nodal program is simple: nodes simply take actions to maximize their expected profit. Nodes do not reason directly about dependencies or consequences of a series of actions, ordering, or the rate at which actions are taken. Because nodes learn the payoff probabilities β_a, they adapt to changing network conditions over time, and different nodes will take different sets of actions depending on their utility functions.

Adjusting prices gives the client of the network control over the behavior of the system, allowing the network to be readily retasked simply by advertising a new price vector. However, because nodes operate to maximize their *expected* profit, an equilibrium arises that balances the actions taken by different nodes in the network. For example, increasing the price of the listen action might substantially reduce the number of nodes that choose to sample or send sensor readings. However, since listening nodes are only paid when other nodes send data, the proportion of sending and listening nodes is kept in balance. This is a valuable aspect of self-scheduling and does not require *explicit* coordination across nodes; this equilibrium arises naturally from the feedback of payments. We demonstrate this aspect of SORA in Section 5.

SORA can be viewed as a general approach to decentralized resource allocation in sensor networks, and is not specifically tailored for data collection and vehicle tracking. However, it is worth keeping in mind that many sensor network applications operate by routing (and possibly aggregating) data towards a single base station, as evidenced by much prior work in this area [5, 11, 17, 26, 46]. It seems clear that the SORA approach could be readily applied to this broad class of systems; for example, SORA could be used to control the execution of query operators

in TinyDB [26].

Extending SORA to other applications involves two basic steps: first, identifying the set of primitive actions and goods that the system should expose, and second, measuring the associated energy costs for each action. For example, exposing a complex operation such as "compute the sum-reduce of sensor readings over a node's k nearest neighborhood" [41] would be straightforward to wrap as an SORA action. One requirement for actions is that data dependencies be made explicit. For example, the send and aggregate actions depend on the sensor reading buffer being non-empty. More complex actions might have a richer set of dependencies that must be met in order to fire. This suggests that nodes should be able to reason about taking a *sequence* of actions to produce some (highly-valued) good; this is another interesting avenue for future work.

5 Experiments and Evaluation

To demonstrate the use of Self-Organizing Resource Allocation in a realistic application setting, we have implemented the SORA-based vehicle tracking system in a sensor network simulator. This simulator captures a great deal of detail, including hardware-level sensor operations and a realistic radio communication model based on traces of packet loss statistics in an actual sensor network. We have also implemented the SORA-based tracking application in TinyOS [16] using the TOSSIM [21] simulator environment. However, due to performance limitations in TOSSIM, the results below are based on our custom simulator that runs roughly an order of magnitude faster. This performance gain is accomplished primarily by eliminating the high overhead associated with the TOSSIM GUI, as well as eliding hardware-level details of node actions that are not relevant to the SORA approach. We have verified that the two simulators produce nearly identical results. The SORA code can be readily ported to run on actual sensor nodes, and we are currently planning to take measurements on our building-wide sensor network testbed [42].

Our evaluation of SORA has three basic goals. First, we show that SORA allows nodes to self-schedule their actions to achieve an efficient allocation of network resources. Second, we show that SORA achieves much greater energy efficiency than traditional scheduling techniques without sacrificing data fidelity. Third, we show that SORA allows the system designer to differentiate node actions by varying energy budgets and price vectors.

We compare the use of SORA to several other implementations of vehicle tracking that use different scheduling techniques. These include the commonly-used static scheduling technique, a dynamic energy-aware scheduling scheme, and a tracking application based on the Berkeley NEST design as described in [43]. These systems are described in detail below.

5.1 Configuration

We simulated a network of 100 nodes distributed semi-irregularly in a 100x100 meter area. The base station (to which all nodes route their messages) is located near the upper-left corner of this area. The energy cost for each action is shown in Figure 1. The simulated vehicle moves in a circular path of radius 30 m at a rate of 1.5 m/sec. Moving the vehicle through such a path causes nodes in different areas of the network to detect the vehicle and route sensor readings towards the base station. The strength of each sensor reading depends on the distance to the vehicle; sensors cannot detect the vehicle when it is more than 11 meters away.

Unless otherwise noted, the energy budget for each node is 1000 J/day, corresponding to a node lifetime of 30.7 days.[1] The prices for all actions were set to an identical value, so nodes have no bias towards any particular action. The exploration probability ϵ is set to 0.05, and the learning parameter α is set to 0.2. We demonstrate the effect of varying these parameters in Section 5.7 and 5.8.

5.2 Comparative Analysis

To compare the use of SORA with more traditional approaches to sensor network scheduling, we implemented three additional versions of the tracking system. The first employs *static scheduling*, in which every node uses a fixed schedule for sampling, aggregating, and transmitting data to the base station. This is the most common approach to designing sensor network applications, typified by fixed sampling periods in TinyDB [26] and directed diffusion [17]. The static schedule is computed based on the energy budget. Given a daily budget of B joules, a node calculates the rate for performing each round of actions (sample, listen, aggregate, transmit, and sleep) in order to meet its budget. For example, given a daily budget of 1000 J, the data collection sequence can be performed once every 0.4 sec. This schedule is conservative, since not all nodes will actually detect the vehicle or transmit data during each period. The same schedule is used for every node in the network, so nodes do not learn which actions they should perform, nor adapt their sampling rate to stimuli such as the approach of the vehicle.

The second approach employs *dynamic scheduling* in which nodes continuously adjust their processing rate based on their current energy budget. In this way, nodes that do not consume energy aggregating or transmitting data can recycle that energy to increase their sampling rate accordingly.

The third and final approach, the *Hoods tracker*, is based on the tracking system implemented using the Hoods communication model [43]. It is largely similar to the dynamically-scheduled tracker, except in the way that nodes calculate the target location. Each node that detects

[1] This assumes that a node runs at 3V with 2850 mA-hours of battery supply.

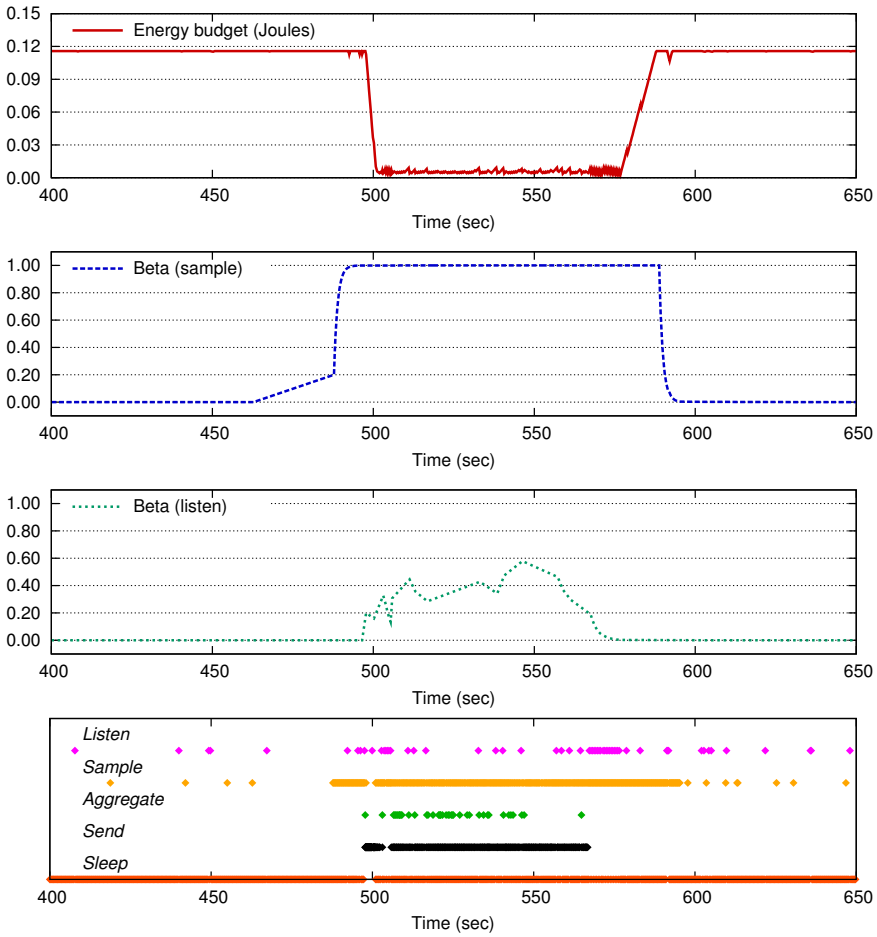


Figure 2: **Actions and energy budget for a single node.** This figure shows the actions taken, the energy budget, and the β values for the listen and sample actions for node 31, which is along the path of the vehicle.

the vehicle broadcasts its sensor reading to its neighbors. The node then listens for some period of time, and if its own reading is the maximum of those it has heard, computes the centroid of the readings (based on the known locations of neighboring nodes) as the estimated target location. This location estimate is then routed towards the base station. We implemented the Hoods tracker to emulate the behavior of a previously-published tracking system for direct comparison with the SORA approach.

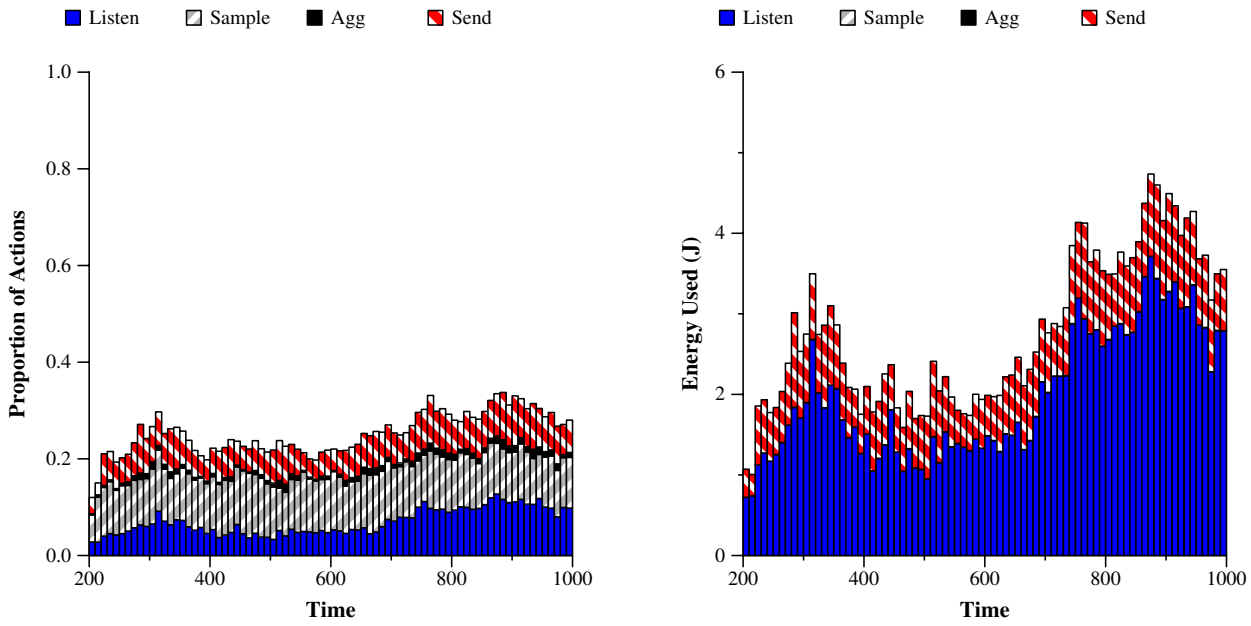
5.3 Agent operation

We begin by demonstrating the operation of the sensor network over time, as nodes learn which actions receive payments. Figure 2 depicts the actions taken, energy budget, and β values for node 31, which is along the path of the vehicle. As the vehicle approaches along its circular path at time $t = 470$, the node determines that it will be paid to sample, aggregate, and send sensor readings. As the vehicle departs around time $t = 590$, the node returns to its original behavior. At certain times (e.g., at $t = 500$ and $t = 548$), the node receives messages from other nodes and routes them towards the base station, explaining the increase in β for the listen action. When the vehicle is not nearby, the node mostly sleeps, since no interesting samples or radio messages are received. The energy bucket fills during this time accordingly; the bucket capacity C is set arbitrarily to 115 mJ, which requires the node to sleep for 20 seconds to fill the bucket entirely.

Observe that the node performs listen and sample actions even when its utility for doing so is low (even zero). This is because the node has enough energy to perform these actions, and the ϵ-greedy action selection policy dictates that it will explore among these alternatives despite negligible utility.

5.4 Network activity over time

Figure 3 shows the proportion of (non-sleeping) actions and energy use by the network over time. As the graph shows, over 60% of the actions taken by nodes during the run are *sleep*. *Listen* and *send* consume far more energy than other actions. The variation in network activity arises due to the movement of the vehicle. For example, at time $t = 600$, the vehicle is closest to the base station, so only those nodes close to the base are sampling and routing data, while the rest of the network is dormant.



(a) Non-sleep actions taken (b) Energy use

Figure 3: **Actions taken and energy use over time.** *This graph shows (a) the proportion of non-sleep actions taken by all nodes in the network and (b) the total energy consumed over time. Over 60% of the actions taken by the network are* sleep.

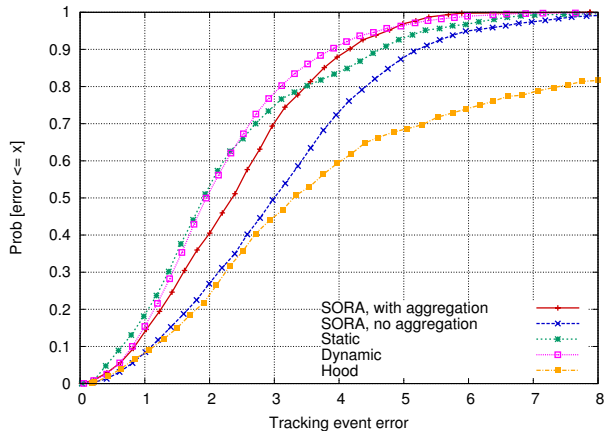


Figure 4: **Tracking accuracy.** *This figure is a CDF of the tracking position error over a run of 1000 sec for each of the tracking systems. The static and dynamic schedulers are the most accurate, since they operate periodically, while SORA has slightly higher error due to its probabilistic operation. Disabling aggregation in SORA causes accuracy to suffer since more readings are delivered to the base station. These three tracking schemes outperform the Hood-based tracker with the same energy budget.*

5.5 Tracking accuracy

To compare SORA with the other scheduling techniques, we are interested in two metrics: tracking accuracy and energy efficiency. We do not expect SORA to be more accurate than the other scheduling approaches, however, it is important that it performs in the same ballpark in order to be viable.

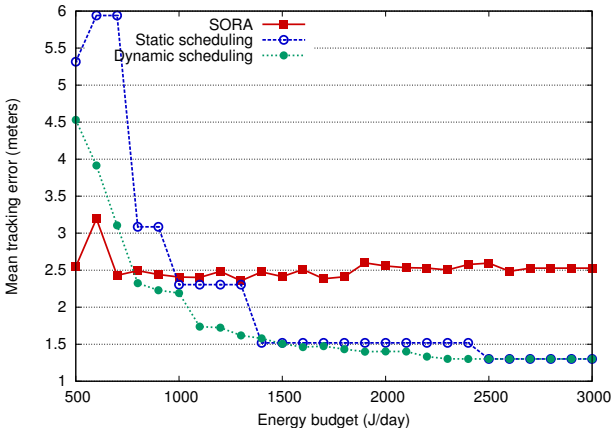
Figure 4 compares the accuracy of the SORA tracker with the other three scheduling techniques. For each position estimate received by the base station, the tracking error is measured as the difference between the estimated and true vehicle position *at the time that the estimate is received*. This implies that position estimate messages that are delayed in the network will increase tracking error, since the vehicle may have moved in the interim. As the figure shows, SORA achieves an 80th percentile tracking error of 3.5 m, only slightly higher than the static and dynamic trackers.

The Hood tracker performs poorly due to its different algorithm for collecting and aggregating sensor data. Figure 5 shows a scatterplot of position estimates received at the base station for each tracking technique. Hood delivers far fewer position estimates and exhibits wider variation in accuracy. Also, disabling aggregation in SORA (by setting the price for the *aggregate* action to 0) causes more position estimates to be delivered that exhibit greater variation than the aggregated samples.

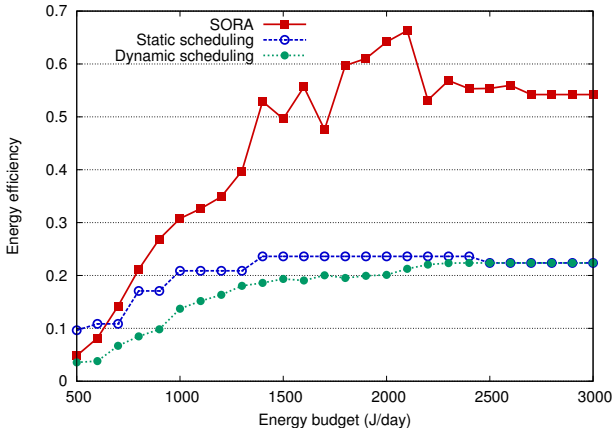
5.6 Energy efficiency

By allowing nodes to self-schedule their operation in response to external stimuli and energy availability, SORA achieves an efficient allocation of energy across the network. For each of the scheduling techniques, we measure the efficiency of resource allocation in terms of the energy cost to acquire each position estimate in proportion to the total amount of "wasted" energy in the network.

For each position estimate received by the base station, we measure the "useful" energy cost of acquiring and routing that data. This includes the sum energy cost of sampling, (optional) aggregation, radio listening, and transmission of the data along each hop. In the case of esti-



(a) Mean tracking error



(b) Energy efficiency

Figure 6: **Tracking accuracy and energy efficiency.** *This figure shows (a) the mean tracking error and (b) overall system energy efficiency as the energy budget is varied.*

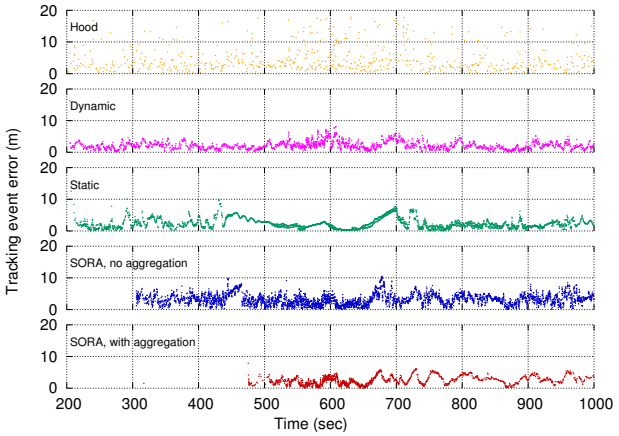


Figure 5: **Tracking accuracy scatterplots.** *These scatterplots show the set of readings delivered to the base station by each tracking system over time. Hood performs poorly and delivers far fewer vehicle position estimates. The effect of disabling aggregation in SORA can be seen clearly.*

mates with aggregated values, we count both the total energy cost for each sensor reading in the estimate, as well as the number of sensor readings represented. Because aggregation amortizes communication overhead across multiple readings, we expect aggregation to reduce the overall per-sample energy cost. The total amount of *useful energy* consumed by the network is the sum of the energy cost for all position estimates produced during a run of the tracking system.

All other energy consumed by the network is *wasted* in the sense that it does not result in data being delivered to the base. In a perfect system, with *a priori* knowledge of the vehicle location and trajectory, communication patterns, and so forth, there would be no wasted energy. In any realistic system, however, there is some amount of waste. For example, nodes may listen for incoming radio messages or take sensor readings that do not result in position estimates. We define *efficiency* as the ratio of the total useful energy consumed by the network to the total energy consumed (useful plus wasted energy).

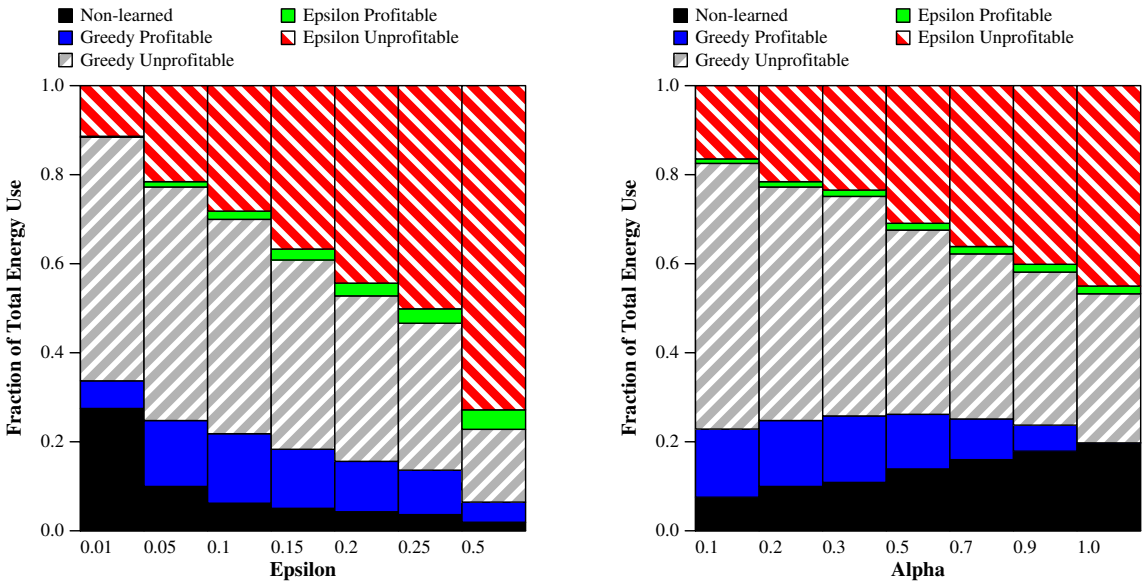
It is important to note that the statically-scheduled and dynamically-scheduled trackers do not make any attempt to save energy beyond their energy budget. Nodes are programmed to operate at a rate that consumes the local energy budget, despite local network conditions. In SORA, however, many nodes may conserve energy by sleeping when they have zero utility for any potential action (e.g., because they are in a quiescent area of the network). The use of reinforcement learning in SORA allows nodes to tune their duty cycle in response to local conditions, significantly extending lifetime.

Figure 6 summarizes the accuracy and efficiency of each scheduling technique as the energy budget is varied. Each system varies in terms of its overall tracking accuracy as well as the amount of energy used. While SORA has a somewhat higher tracking error compared to the other scheduling techniques, it demonstrates the highest efficiency, exceeding 66% for an energy budget of 2100 J. The static and dynamic schedulers achieve an efficiency of only 22%. In SORA, most nodes use far less energy than the budget allows. The ability of SORA to "learn" the duty cycle on a per-node basis is a significant advantage for increasing network lifetimes.

5.7 Varying learning parameters

Apart from the energy budgets and prices, two parameters that strongly affect node behavior in SORA are ϵ, the exploration probability, and α, the EWMA gain for learning action success probabilities. By varying ϵ, we can trade off increased energy waste (for exploring the action space) for faster response to changing network conditions. By varying α, the system reacts more or less quickly to changes in success probabilities; higher values of α cause a node to bias action selection towards more-recently profitable actions.

Figure 7(a) shows the effect of varying ϵ from 0.01 to 0.5. As the probability of taking a random action increases,



(a) Varying exploration probability ϵ **(b) Varying EWMA filter gain α**

Figure 7: **Effect of varying exploration and learning parameters.** *(a) α is held constant at 0.2 and the probability of taking a random action ϵ is varied. (b) ϵ is held constant at 0.05 and the EWMA filter gain is α is varied.*

the proportion of energy wasted taking those actions also increases. However, the proportion of energy wasted taking the "greedy" action (the action with the highest expected probability of success) decreases, since nodes learn more rapidly which actions are profitable by exploring the action space.

Figure 7(b) shows a similar result for varying α. When α is increased, nodes react very quickly to changes in action success. When $\alpha = 1.0$, if an action is successful once, the node will immediately prefer it over all others. Likewise, the node will immediately ignore a potentially profitable action the first time it is unsuccessful. As a result, the proportion of energy used on successfully choosing the greedy action decreases. Also, since the node's action selection policy is increasingly myopic, nodes spend more time sleeping. As a result, a greater proportion of energy is spent on exploratory actions since few "greedy" actions are considered worthwhile.

5.8 Heterogeneous energy budgets and prices

SORA allows nodes to be differentiated with respect to their energy budgets, as well as the prices under which they operate. For example, certain nodes may have access to a large power supply and should be able to perform more power-hungry operations than nodes operating off of small batteries. Likewise, advertising different price vectors to different nodes allows them to be customized to take certain actions.

Figure 8 shows the behavior of the tracking system where 20% of the nodes are given a large energy budget of 3000 J/day, effectively allowing them to ignore energy constraints for the purpose of selecting actions. The large energy budget nodes automatically elect to perform a greater number of listen and send actions, while the other nodes mostly perform sample actions, which consume far less energy overall. Identical prices are used throughput the network, showing that differences in energy budget have a profound effect on resource allocation.

Advertising different price vectors to different sets of nodes is another way to specialize behavior in SORA. Figure 9 shows a case where 20% of the nodes are configured as "routers" with all prices set to 0, except for listening, aggregation, and sending. The other nodes act as "sensors" with nonzero prices only for sampling and sending. As the figure shows, each group of nodes exhibits very different behavior over the run, with sensor nodes performing a large number of sampling and send actions, while router nodes primarily listen and transmit. Routers spend a great deal of time sleeping because most actions (e.g., aggregation and sending) are unavailable, and listening consumes too much energy to perform continually.

6 Future Work and Conclusions

The design of sensor network applications is complicated by the extreme resource limitations of nodes and the unknown, often time-varying, conditions under which they operate. Current approaches to resource management are often extremely low-level, requiring that the operation of individual sensor nodes be specified manually. In this paper, we have presented an technique for resource allocation in sensor networks in which nodes act as self-interested agents that select actions to maximize profit, subject to energy limitations. Nodes self-schedule their local actions in response to feedback in the form of payments. This allows nodes to adapt to changing conditions and specialize their behavior according to physical location, routing topology,

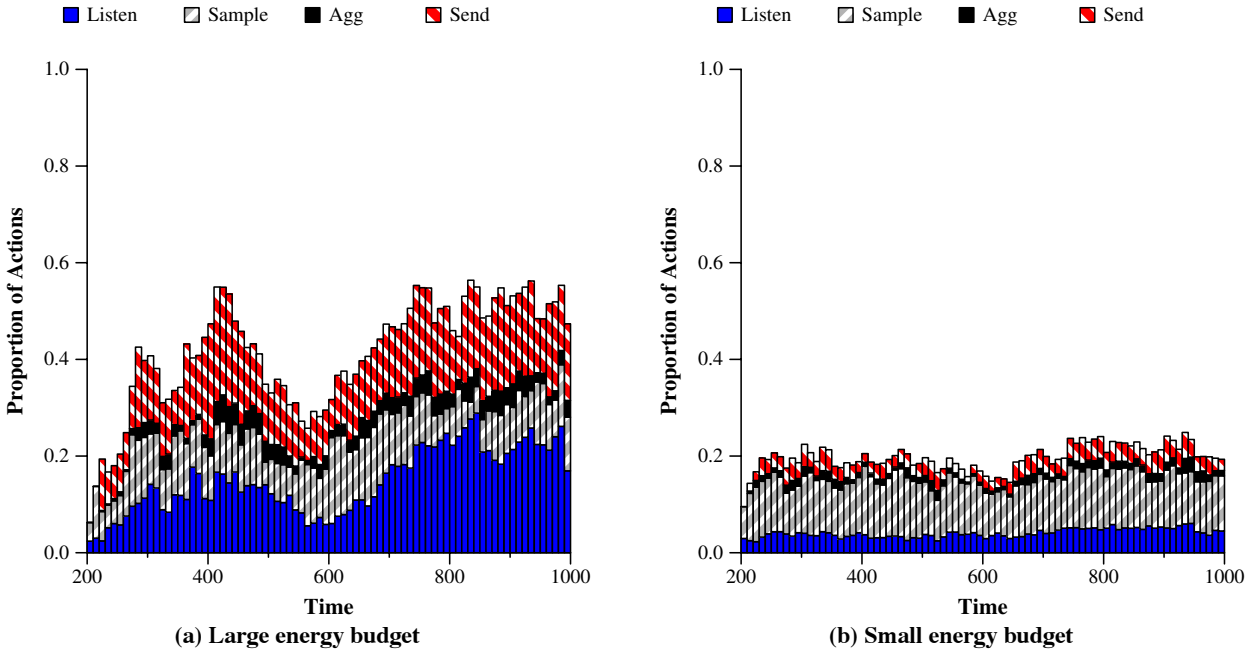


Figure 8: **Exploiting heterogeneous energy budgets.** *Here, 20% of the nodes are given a large energy budget of 3000 J/day (a), where the rest of the nodes use a smaller energy budget of 500 J/day (b). The large energy budget nodes automatically take on a greater proportion of the energy load in the system, choosing to perform a far greater number of listen and send actions than the low-energy nodes.*

and energy reserves.

Exploiting techniques from reinforcement learning and economic theory yields new insights into the allocation of scarce resources in an adaptive, decentralized fashion. Our initial work on SORA raises a number of interesting questions that we wish to explore in future work. These are described in summary below.

Equilibrium pricing: As discussed earlier, a system is in competitive equilibrium (Pareto optimal) when prices are selected such that supply of goods equals demand. This is an attractive model for allowing multiple users to allocate and share sensor network resources in an optimal fashion. However, such an approach raises a number of practical problems that must be addressed before it can be applied to sensor networks. The traditional tâtonnement approach [29] is to increase prices on undersupplied goods (and vice versa for oversupplied goods) until reaching equilibrium, and execute trade only after prices have been selected. A real system must depart from this approach in that it operates continuously. As a result, since supply and demand may lag price adjustments, true equilibrium may never be reached.

In addition, calculating equilibrium prices generally requires clients to have *global* information on the supply provided by each sensor node at the currently-proposed prices. We are exploring techniques in which aggregate supply information is collected and piggybacked on other transmissions to the base station. However, clients must then operate on incomplete and out-of-date supply information. Another approach is to collect supply information at several price points simultaneously, allowing the client to adjust prices based on the resulting gradient information.

Richer pricing models: More complex pricing schemes can be used to induce sophisticated behaviors in the network. For example, rather than pricing only those goods that result from single actions, we can price *sequences* of actions. Consider aggregating multiple sensor readings into a single value for transmission. Rather than price the final aggregate value and requiring an agent to reason about a sequence of actions to achieve that result, we can establish prices for each step in the sequence and introduce control or data dependencies between actions. Another question is that of *location-based prices*, in which goods are priced differently in different areas of the network. This can be used to establish a ring of "sentry nodes" around the perimeter of the network that wake other nodes in the interior when the entrance of a vehicle is detected.

New application domains: We intend to explore other applications for the SORA technique. As discussed earlier, this requires that nodes be programmed with new actions and corresponding energy consumption models. One application that we are actively investigating involves sensor networks for emergency medical care and disaster response [25]. This scenario involves establishing multicast communication pathways between multiple vital sign sensors worn by patients and handheld devices carried by rescuers and doctors. We envision SORA providing a mechanism for efficient bandwidth and energy allocation in this environment.

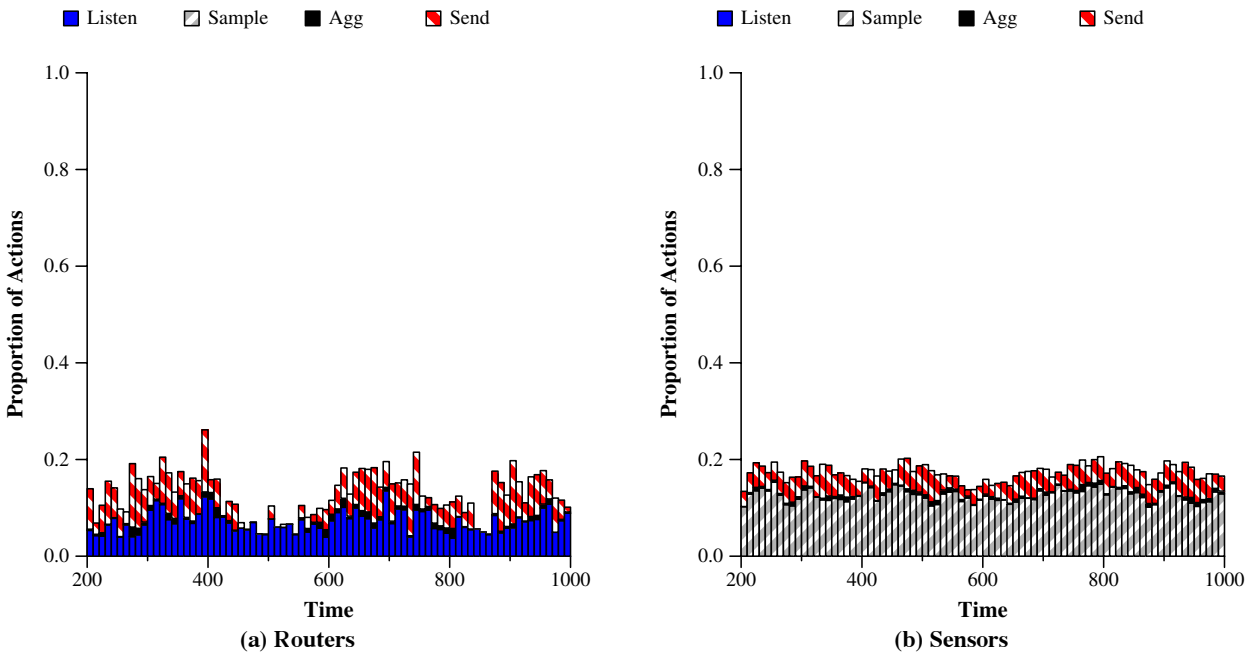


Figure 9: **Specialization through pricing.** *Here, 20% of the nodes are configured as "routers" (left) using prices for listening, aggregation, and sending. The other 80% of the nodes are configured as "sensors" (right) and only have prices for sampling and sending. As the figure shows the proportion of actions taken by each group of nodes differs greatly according to the prices.*

Integration with programming languages: Finally, our broader research agenda for sensor networks involves developing high-level *macroprogramming* languages that compile down to local behaviors of individual nodes. SORA presents a suite of techniques for scheduling node actions and managing energy that could be integrated into such a language. For example, TinyDB's SQL-based query language could be implemented using SORA to control the execution of query operators on each node, rather than the current model of relying on a static schedule. We have completed the initial design of a functional macroprogramming language for sensor networks that compiles down to a simple per-node state machine that could be readily implemented using a SORA-based model [31, 32].

Acknowledgments

The authors wish to thank our shepherd, Amin Vahdat, as well as the anonymous reviewers for their comments on this paper.

References

[1] S. Adlakha, S. Ganeriwal, C. Schurgers, and M. B. Srivastava. Density, accuracy, latency and lifetime tradeoffs in wireless sensor networks - a multidimensional design perspective. In review, 2003.

[2] L. Anderegg and S. Eidenbenz. Ad hoc-VCG: a truthful and cost-efficient routing protocol for mobile ad hoc networks with selfish agents. In *Proc. MOBICOM'03*, September 2003.

[3] A. AuYoung, A. C. Snoeren, A. Vahdat, and B. Chun. Resource allocation in federated distributed computing infrastructures. In *Proc. First Workshop on Operating System and Architectural Support for the on demand IT InfraStructure*, October 2004.

[4] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-based load management in federated distributed systems. In *Proc. First Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.

[5] A. Boulis, S. Ganeriwal, and M. B. Srivastava. Aggregation in sensor networks: An energy - accuracy tradeoff. In *Proc. IEEE workshop on Sensor Network Protocols and Applications*, 2003.

[6] R. Brooks, P. Ramanathan, and A. Sayeed. Distributed target classification and tracking in sensor networks. *Proceedings of the IEEE*, November 2003.

[7] R. Buyya. *Economic-based Distributed Resource Management and Scheduling for Grid Computing*. PhD thesis, Monash University, April 2002.

[8] Center for Information Technology Research in the Interest of Society. Smart buildings admit their faults. `http://www.citris.berkeley.edu/applications/disaster_response/smartbuil%dings.html`, 2002.

[9] A. Cerpa, J. Elson, D. Estrin, L. Girod, M. Hamilton, and J. Zhao. Habitat monitoring: Application driver for wireless communications technology. In *Proc. the Workshop on Data Communications in Latin America and the Caribbean*, Apr. 2001.

[10] S. H. Clearwater, editor. *Market-Based Control: A Paradigm for Distributed Resource Allocation*. World Scientific, 1996.

[11] D. Ganesan, B. Greenstein, D. Perelyubskiy, D. Estrin, and J. Heidemann. An evaluation of multi-resolution search and storage in resource-constrained sensor networks. In *Proc. the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, November 2003.

[12] J. S. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building efficient wireless sensor networks with low-level naming. In *Proc. the 18th SOSP*, Banff, Canada, October 2001.

[13] W. Heinzelman, A. Chandrakasan, and H. Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *Proc. the 33rd Hawaii International Conference on System Sciences (HICSS)*, January 2000.

[14] W. Heinzelman, J. Kulik, and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *Proc. the 5th ACM/IEEE Mobicom Conference*, August 1999.

[15] J. M. Hellerstein, W. Hong, S. Madden, and K. Stanek. Beyond average: Towards sophisticated sensing with queries. In *Proc. the 2nd International Workshop on Information Processing in Sensor Networks (IPSN '03)*, March 2003.

[16] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In

Proc. the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 93–104, Boston, MA, USA, Nov. 2000.

[17] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proc. International Conference on Mobile Computing and Networking*, Aug. 2000.

[18] L. Kang, S. Lahaie, G. Mainland, D. C. Parkes, and M. Welsh. Using virtual markets to program global behavior in sensor networks. In *Proc. 11th ACM SIGOPS European Workshop*, Leuven, Belgium, September 2004.

[19] B. Karp and H. T. Kung. GPSR: Greedy perimeter stateless routing for wireless networks. In *Proc. the Sixth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 2000)*, Boston, MA, August 2000.

[20] V. A. Kottapalli, A. S. Kiremidjian, J. P. Lynch, E. Carryer, T. W. Kenny, K. H. Law, and Y. Lei. Two-tiered wireless sensor network architecture for structural health monitoring. In *Proc. the SPIE 10th Annual International Symposium on Smart Structures and Materials*, San Diego, CA, March 2000.

[21] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proc. the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, November 2003.

[22] P. Levis, N. Patel, S. Shenker, and D. Culler. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proc. the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[23] D. Li, K. Wong, Y. H. Hu, and A. Sayeed. Detection, classification and tracking of targets in distributed sensor networks. *IEEE Signal Processing Magazine*, 19(2), March 2002.

[24] J. Liu, P. Cheung, L. Guibas, and F. Zhao. A dual-space approach to tracking and sensor management in wireless sensor networks. In *Proc. First ACM International Workshop on Wireless Sensor Networks and Applications*, September 2002.

[25] K. Lorincz, D. Malan, T. R. F. Fulford-Jones, A. Nawoj, A. Clavel, V. Shnayder, G. Mainland, S. Moulton, and M. Welsh. Sensor Networks for Emergency Response: Challenges and Opportunities. *IEEE Pervasive Computing*, Oct-Dec 2004.

[26] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proc. the 5th OSDI*, December 2002.

[27] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proc. the ACM SIGMOD 2003 Conference*, June 2003.

[28] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, Atlanta, GA, USA, Sept. 2002.

[29] A. Mas-Colell, M. D. Whinston, and J. R. Green. *Microeconomic Theory*. Oxford University Press, 1995.

[30] T. Mullen and M. P. Wellman. Some issues in the design of market-oriented agents. In W. et al., editor, *Intelligent Agents: Theories, Architectures and Languages*, volume 2. Springer-Verlag, 1996.

[31] R. Newton, Arvind, and M. Welsh. Building up to macroprogramming: An intermediate language for sensor networks. In *Proc. Fourth International Conference on Information Processing in Sensor Networks (IPSN'05)*, April 2005.

[32] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. In *Proc. the First International Workshop on Data Management for Sensor Networks (DMSN)*, Toronto, Canada, August 2004.

[33] S. Rhee and S. Liu. An ultra-low power, self-organizing wireless network and its applications to noninvasive biomedical instrumentation. *Proc. IEEE/Sarnoff Symposium on Advances in Wired and Wireless Communications*, March 2002.

[34] C. Schurgers, V. Tsiatsis, S. Ganeriwal, and M. Srivastava. Topology management for sensor networks: Exploiting latency and density. In *Proc. MobiHoc*, 2002.

[35] S. Shenker. Fundamental design issues for the future Internet. *IEEE Journal on Selected Areas in Communications*, 13(7), September 1995.

[36] M. Stonebraker, R. Devine, M. Kornacker, W. Litwin, A. Pfeffer, A. Sah, and C. Staelin. An economic paradigm for query processing and data migration in Mariposa. In *Proc. the 3rd International Conference on Parallel and Distributed Information Systems*, September 1994.

[37] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[38] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrating communication and computation. In *Proc. the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[39] C. A. Waldspruger, T. Hogg, B. A. Huberman, J. O. Kephart, and S. Stornetta. Spawn: A Distributed Computational Economy. *IEEE Transactions on Software Engineering*, 18(2):103–177, February 1992.

[40] M. P. Wellman. Market-oriented programming: Some early lessons. In S. Clearwater, editor, *Market-Based Control: A Paradigm for Distributed Resource Allocation*. World Scientific, 1996.

[41] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *Proc. the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.

[42] G. Werner-Allen, P. Swieskowski, and M. Welsh. Motelab: A wireless sensor network testbed. In *Proc. Fourth International Conference on Information Processing in Sensor Networks (IPSN'05), Special Track on Platform Tools and Design Methods for Network Embedded Sensors (SPOTS)*, April 2005.

[43] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: A neighborhood abstraction for sensor networks. In *Proc. the International Conference on Mobile Systems, Applications, and Services (MOBISYS '04)*, June 2004.

[44] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proc. the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, November 2003.

[45] Y. Xu and W.-C. Lee. On localized prediction for power efficient object tracking in sensor networks. In *Proc. 1st International Workshop on Mobile Distributed Computing*, May 2003.

[46] Y. Yao and J. E. Gehrke. The Cougar approach to in-network query processing in sensor networks. *ACM Sigmod Record*, 31(3), September 2002.

[47] F. Zhao, J. Liu, J. Liu, L. Guibas, and J. Reich. Collaborative signal and information processing: An information directed approach. *Proc. the IEEE*, 91(8):1199–1209, 2003.

Beacon Vector Routing: Scalable Point-to-Point Routing in Wireless Sensornets

Rodrigo Fonseca* Sylvia Ratnasamy† Jerry Zhao‡ Cheng Tien Ee*
David Culler* Scott Shenker*,‡ Ion Stoica*

*University of California, Berkeley {rfonseca, ct-ee, culler, istoica}@cs.berkeley.edu
†Intel Research, Berkeley sylvia@intel-research.net
‡International Computer Science Institute {zhao, shenker}@icsi.berkeley.edu

Abstract

We propose a practical and scalable technique for point-to-point routing in wireless sensornets. This method, called Beacon Vector Routing *(BVR), assigns coordinates to nodes based on the vector of hop count distances to a small set of beacons, and then defines a distance metric on these coordinates. BVR routes packets greedily, forwarding to the next hop that is the closest (according to this beacon vector distance metric) to the destination. We evaluate this approach through a combination of high-level simulation to investigate scaling and design tradeoffs, and a prototype implementation over real testbeds as a necessary reality check.*

1 Introduction

The first generation of sensornet deployments focused primarily on data collection [23, 9]. In support of this task, most current sensornet code bases [11, 7] offer only the basic tree-based many-to-one and one-to-many routing primitives; protocols such as Directed Diffusion [12], TAG[21], and others build trees that can both broadcast commands and collect data, with various forms of aggregation along the collection path. However, a growing number of recent proposed uses require more sophisticated point-to-point routing support. These include applications such as PEG (a pursuer-evader game in which a large network tracks the movement of evader robots [4]), approaches such as reactive tasking (commands based on local sensing results), and data query methods such as multi-dimensional range queries [20], spatial range queries[8], and multi-resolution queries [6], and data-centric storage [29].

Unfortunately, it is hard to test these ideas because there is currently no practical and broadly-applicable implementation of point-to-point routing for sensornets. We know of two implementations of a reduced AODV and of GPSR [15], but they haven't been reported on in the literature. As we discuss in the next section, they have limitations on their applicability. It isn't clear how important these newly proposed uses are, but without a point-to-point routing protocol we will never be able to evaluate their true utility. Moreover, the applications and services that emerge from the sensornet community will depend, in part, on which routing primitives have scalable and practical implementations; hence, the lack of a robust implementation of point-to-point routing might well limit the scope of future sensornet applications.

The lack of point-to-point implementations is in stark contrast with the bevy of proposed designs in this space. As we review in the next section, there have been many different approaches to this problem, but none has resulted in a reliable implementation. We speculate that this is largely due to the clash between the complexity of these proposals and the demanding requirements of sensornet implementation. Sensornet implementations should not only meet stringent scaling, robustness, and energy efficiency standards, but they should also function on a hardware base that has severe resource limitations (in terms of memory and packet length) and varying quality radios. The impact of these factors on design is best illustrated by the experience of the TinyOS developers (described in [19]) where the algorithmically trivial flooding and tree construction primitives took three years and five successive implementations to get right.

Thus, simplicity is our primary design requirement. We make minimal assumptions about radio quality, presence of GPS, and other factors, and want minimal complexity in the algorithm itself. We do so by using the previous hard-won successes in tree-building as the basic building block of our more general routing protocol. We select a few *beacon* nodes and construct trees from them to every other node (using standard techniques). As a result, every node is aware of its distance (in hops) to every beacon and these beacon vectors can serve as coordinates. After defining a distance metric over these coordinates, we can use a simple greedy distance-minimizing routing algorithm. This approach, which we call Beacon Vector Routing (BVR), requires very little state, overhead, or pre-configured information (such as geographic location of nodes). Routes are based on connectivity, which nodes are naturally aware of, and in our simula-

tions and measurements appear to be reasonably close to the minimal distance paths.

The remainder of this paper is organized as follows: we compare BVR to related work in Section 2, trying to illustrate both qualitative and quantitative differences between the various proposals. We present the BVR routing algorithm in Section 3 and use high-level simulations to investigate scaling and design tradeoffs in Section 4. We describe the details of the BVR implementation in Section 5 and evaluate our implementation with two independent testbeds (to provide a much-needed reality check on our results) in Section 6. Future directions and our conclusions are presented in section 7.

2 Related Work

The many proposals for point-to-point routing in ad-hoc wireless networks [26] can be broadly divided into four very different categories. We discuss each of these in turn, highlighting their pros and cons when applied to sensor networks, and then contrast them with BVR. Table 1 summarizes our discussion.

Shortest Path: This is the classical approach to routing in which a distributed form of Dijkstra's algorithm is used to compute the shortest path between a source and destination. In early protocols such as Distance-Vector, Link-State and DSDV, the shortest path between all possible source-destination pairs is computed and every node stores its next hop to every destination. For a network with n nodes, this results in O(n^2) message exchanges for route discovery and O(n) routing state at each node. This overhead, particularly the per-node state, scales poorly to large networks. For example, a Mica2 mote has only 4KB RAM; in a 1000 node network, a node's routing table alone would exhaust this.

To reduce this overhead, Johnson *et al.* proposed the use of *on-demand* route discovery [13]. The resulting improvement in scalability depends entirely on the overall traffic pattern and, while these protocols perform admirably in many settings, they are not well-suited to cases with traffic between many source-destination pairs (which can be expected in DIM [20], PEG [4], Dimensions [6], *etc.*).

Hierarchical Addressing: The (wired) Internet uses careful address allocation which allows significant route aggregation and thus smaller routing tables. This is infeasible in sensor networks in part because of the overhead of manual configuration but also because a sensornet's connectivity graph is dependent on the details of its physical environment and is often quite variable; this makes it difficult to determine *a priori* how addresses should be assigned.

Francis' [31] elegant Landmark Routing (LR) proposal solves this problem by allowing nodes to self–configure their addresses. LR uses a hierarchical set of landmark nodes that periodically send scoped route discovery messages. A node's address is the concatenation of its closest landmark at each level in the hierarchy. LR reduces the overhead of route setup to O($n \log n$) and nodes only hold state for their immediate neighbors and their next hop to each landmark. However, LR requires a protocol that creates and maintains this hierarchy of landmarks and appropriately tunes the landmark scopes. The original LR proposal does not address the details of such a protocol and no workable implementation has been deployed. More recent proposals adopting this approach have been fairly complex [17], in conflict with our design goal of configuration simplicity.

Geographic Coordinates: A different and potentially attractive solution for sensor networks is based on geographic routing [15, 1, 16]. Here, nodes are identified by their geographic coordinates and routing is done greedily; at each step, nodes pick as next-hop the neighbor that is closest to the destination. When a node has no neighbor that is closer to the destination, these protocols enter perimeter mode, where the right-hand rule is used to forward a packet along a planarized subgraph until it reaches a node closer to the destination than the starting point of perimeter mode (then it resumes its greedy forwarding). Geographic routing is eminently scalable— it incurs $O(1)$ overhead for route discovery and $O(1)$ routing tables (a node need only discover and store its one-hop neighbors), the planarization techniques are purely local, and path lengths are close to the shortest path [15]. Unfortunately, geographic routing has two problems. First, the correctness of the common (local) planarization algorithms, and hence the correctness of perimeter mode routing, relies on a unit-graph assumption under which a node hears all transmissions from nodes within its fixed radio range and never hears transmissions from nodes outside this range. Measurement studies [32, 33, 3] have shown that this assumption is grossly violated by real radios. Second, and more seriously, such routing requires that each node know its geographic coordinates. While there are some sensor nodes that are equipped with GPS, the most widely used node, the Berkeley mote [10], is not. Moreover, even when available, GPS does not work in certain physical environments and the various proposed localization algorithms [27] are not precise enough (at least not in all settings) to be used for geographic routing. Finally, even ignoring all the above, greedy geographic may be substantially suboptimal because it does not use real connectivity information and geography is, as we show in Section 6, not always in congruence with true network connectivity (*e.g.*, in the face of obstacles or consistent interference).

Virtual Coordinates: Motivated by the ideal scaling properties of schemes like GPSR, two recent proposals

Metric	DV/LS	AODV/DSR	Landmark	GPSR	NoGeo	GEM	BVR
Setup overhead	O(n^2)	n/a	O($nlogn$)	—	O($n\sqrt{n}$)	O(dn)	O(rn)
Route overhead	1	O(n) if route is uncached; else O(1)	$>$ O(1)	O(1)	O(1)	O(1)	O(1)
Per-node State	O(n)	depends on traffic pattern	O($d + logn$)	O(d)	O(d)	O(d)	O($d + r$)
Delivery Guar.	yes	yes	yes	assuming unit graph topologies	yes	yes	yes
Configuration	no	no	hierarchically scoped landmarks	geographic positions	perimeter discovery	coordinate construction	pick r random nodes
Local recovery	? (limited)	no	? (limited)	yes	yes	yes	yes
Local repair	no	no	no	yes	mostly	no	no

Table 1: **Design considerations for sensornet routing algorithms and the tradeoffs using current solutions. This table is intended to be illustrative rather than definitive.** *n is the number of nodes. Setup overhead refers to the total message traffic generated to setup pairwise routes. While to some extent a one-time cost, this is also indicative of the overhead incurred by topology changes. Route overhead refers to the number of transmissions relative to the optimal shortest path. Per-node state is the number of routing entries maintained at each node. Delivery guarantee indicates whether a solution guarantees, at the* algorithmic *level, whether the protocol is guaranteed to find a route to all destinations. Local recovery refers to a node's ability to route around failed nodes in the absence of* any *recovery protocol. Local repair refers to a protocol's ability to limit the impact of node failure to that node's immediate neighbors. d denotes a node's degree (immediate neighbors) and r denotes the number of beacons in BVR, typically a small constant (10). A question mark indicates that our uncertainty about the claimed performance as the literature may not have addressed the relevant consideration.*

attempt to use geographic routing ideas without requiring geographic coordinates. The NoGeo scheme [25] creates synthetic coordinates through an iterative relaxation algorithm that embeds nodes in a Cartesian space. The initialization technique for this scheme requires roughly $O(\sqrt{n})$ nodes to flood the network, and for each of these flooding nodes to store the entire $O(\sqrt{n} \times \sqrt{n})$ matrix of distances (in hops). This is keeping $O(n)$ state at roughly $O(\sqrt{n})$ nodes, an impractical burden in large networks. GEM [24] uses a more scalable initialization scheme but employs an intricate recovery process in which, when nodes fail or radio links degrade, a potentially large number of nodes in the system must recompute routing labels so as to maintain GEM's regular topological structure. Neither NoGeo nor GEM have been implemented on any hardware platform and, while they represent significant conceptual advances, the complexity of coordinate construction and maintenance in these schemes is likely to render both quite difficult to implement and operate in practice.

BVR: BVR borrows, and differs, from each of the above. BVR incurs smaller routing state than the shortest path algorithms (constant vs. O(n)). From Landmark Routing, BVR borrows from the notion of using landmarks to infer node addresses, though the details of the addressing and forwarding are entirely different. Moreover, BVR's beacons are randomly chosen and need not adhere to any particular structure. BVR uses greedy forwarding over node coordinates, but (unlike GPSR) does not require geographic information, makes no assumptions about radio connectivity, and (unlike NoGeo and GEM) uses a very simple coordinate construction algorithm. As mentioned earlier, the core mechanism in BVR is the construction of reverse path trees similar to [22, 34]. However, unlike these schemes, BVR does not directly route along these trees and instead supports point-to-point communication.

We have recently become aware of Logical Coordinate Routing [2], developed simultaneously and independently of BVR, which employs the same idea of nodes obtaining coordinates from a set of landmarks and routing to minimize a distance function on these coordinates. The main difference is the alternative method of routing when local minima are reached in greedy routing: they backtrack the packet along the path, until a suitable path is found, or the route fails at the origin. While they never have to resort to small scoped floods, as does BVR, their algorithm does require that the nodes keep a record of all packets forwarded recently, increasing the amount of state in the nodes.

3 The BVR Algorithm

BVR defines a set of coordinates and a distance function to enable scalable greedy forwarding. These coordinates are defined in reference to a set of "beacons" which are a small set of randomly chosen nodes; using a fairly standard reverse path tree construction algorithm every node learns its distance, in hops, to each of the beacons. A node's coordinates is a vector of these distances. On the occasion that greedy routing with these coordinates fails, we use a correction mechanism that guarantees delivery.

Let q_i denote the distance in hops from node q to beacon i. Let r denote the total number of beacon nodes. We define a node q's position $\mathcal{P}(q)$ as being the vector of these beacon distances: $\mathcal{P}(q) = \langle q_1, q_2, \cdots, q_r \rangle$. Two nodes can have the same coordinates, so we always retain a node identifier to disambiguate nodes in such cases. Nodes must know the positions of their neighbors

Packet fields	Description
$pkt.dst$	the destination's unique identifier
$pkt.\mathcal{P}(dst)$	destination's BVR position
$pkt.\overline{\delta^{min}}$	δ_i^{min} seen, $i \in 1, ..., k$

Table 2: BVR packet header fields

to make routing decisions, so nodes periodically send a local broadcast messages announcing their coordinates.

To route, we need a distance function $\delta(p, d)$ on these vectors that measures how good p would be as a next hop to reach a destination d. The goal is to pick a function so that using it to route greedily usually results in successful packet delivery. The metric should favor neighbors whose coordinates are more similar to the destination. Minimizing the absolute difference component-wise is the simplest such metric. The key piece of intuition driving our design is that it is more important to move *towards* beacons than to move *away* from beacons. When trying to match the destination's coordinates, we move towards a beacon when the destination is closer to the beacon than the current node; we move away from a beacon when the destination is further from the beacon than the current node. Moving towards beacons is always moving in the right direction, while moving away from a beacon might be going in the wrong direction (in that the destination might be on the other side of the beacon). To embody this intuition, we use the following two sums:

$$\delta_k^+(p, d) = \sum_{i \in C_k(d)} \max(p_i - d_i, 0) \quad \text{and}$$

$$\delta_k^-(p, d) = \sum_{i \in C_k(d)} \max(d_i - p_i, 0),$$

where $C_k(d)$ is the set of the k closest beacons to d. δ_k^+ is the sum of the differences for the beacons that are closer to the destination d than to the current routing node p, while δ_k^- measures the sum of the distances to the farther beacons. We choose the next hop that minimizes δ_k^+ and, when there is a tie, we break it by minimizing δ_k^-. In practice, we implement this by minimizing the sum $\delta_k = A\delta_k^+ + \delta_k^-$ for some sufficiently large constant A. In our implementation we use $A = 10$. In addition, δ_k only considers the k closest beacons to d. This serves to reduce the number of distance elements d_i that must be carried in the packet, and is consistent with the idea of moving towards close beacons.

To route to a destination dst, a packet has three header fields, summarized in Table 2: (1) the destination's unique identifier, (2) it's position $\mathcal{P}(dst)$ defined over the beacons in $C_k(dst)$, and (3) $\overline{\delta_{min}}$, a k-position vector where δ_i^{min} is the minimum δ that the packet has seen so far using $C_i(dst)$, the i closest beacons to dst. δ_i^{min} can guarantee that the route will never loop.

Algorithm 1 lists the pseudo-code for BVR forwarding. The parameters are r, the total number of beacons, and $k \leq r$, the number of beacons that define a destination's position. Forwarding a message starts with a greedy search for a neighbor that improves the minimum distance we have seen so far. When forwarding the message, the current node (denoted $curr$) chooses among its neighbors the node $next$ that minimizes the distance to the destination. We start using the k closest beacons to the destination, and if there is no improvement, we successively drop beacons from the calculation.

Algorithm 1 BVR forwarding algorithm

```
BVR_FORWARD(node curr, packet P)
  // first update packet header
  for (i = 1 to k) do
    P.δ_i^min = min(P.δ_i^min, δ_i(curr, P.dst))

  // try greedy forwarding first
  for (i = k to 1) do
    next ← argmin_{x∈NBR(curr)}{δ_i(x, P.dst)}
    if (δ_i(next, P.dst) < P.δ_i^min ) then
      unicast P to next

  //greedy failed, use fallback mode
  fallback_bcn ← closest beacon to P.dst
  if (fallback_bcn != curr) then
    unicast P to PARENT(fallback_bcn)

  //fallback failed, do scoped flood
  broadcast P with scope P.P(dst)[fallback_bcn]
```

In some situations greedy forward may fail, in that no neighbor will improve on δ_i^{min} for any i. We use a 'fallback' mode to correct this. The intuition behind fallback mode is that if a node cannot make progress towards the destination itself, it can instead forward towards a node that it knows is close to the destination and towards which it does know how to make progress. The node forwards the packet towards the beacon closest to the destination; *i.e.*, to its parent in the corresponding beacon tree. The parent will forward as usual – first trying to forward greedily and, failing to do so, using fallback mode.

A packet may ultimately reach the beacon closest to the destination and still not be able to make greedy progress. At this point, the root beacon initiates a scoped flood to find the destination. Notice that the required scope of the flood can be precisely determined – the distance in hops from the flooding beacon to the destination is determined from the destination's position in the packet header. While this ensures that packets can always reach their destination, flooding is an inherently expensive operation and hence we want to minimize the frequency with which it is performed, and also its scope. Our results show both these numbers to be low.

Beacon Maintenance Sensor network nodes are prone to failure and we must provide a mechanism to maintain

the set of beacons when they fail. We first note that the algorithm we described can function with fewer than r beacons, and even when there is inconsistency in the beacon sets nodes are aware of, by routing only based on the beacons they have in common. Thus, the beacon maintenance need not be perfect, it only needs to guide the system towards a state where there are r globally recognized beacons. We now sketch such an algorithm. For convenience, we describe the simplest algorithm we've used; we've also experimented with more advanced algorithms but describing them would take us too far afield.

To detect beacon failures, each entry in the beacon vector is associated with a sequence number. Beacons periodically advance their sequence number; if a node detects that a sequence number has not been updated with a given timeout period then it deletes that beacon from the set it uses to route (note that this decision need not be globally consistent). When the number of beacons alive falls below a configurable parameter r, non-beacon nodes will nominate themselves as beacons. Using ideas from SRM [5], each node sets a timer that is a function of its unique identifier and, when the timer expires, it starts acting like a beacon. If it detects that there are more than r beacons with identifiers smaller than its identifier, it then ceases to be a beacon. Algorithm 2 shows our beacon selection algorithm. More sophisticated beacon maintenance protocols can be designed to more fully optimize beacon placement and suppression.

Algorithm 2 BVR beacon maintenance algorithm

BEACON_ELECT_MYSELF(r, B)
  // invoked periodically; B is the current set of beacons
  if ($|B| > r$) **then**
    set timer $T = \frac{log(myID)}{\log(maxID(B))} * T_{max} + jitter$

TIMER_T_EXPIRES(r, B)
  if ($|B| < r$) **then**
    Announce myself as a beacon

BEACON_SUPPRESS_MYSELF(r, B)
  if ($myID \in B$)&($|B| > r$)&($myID > r^{th}$_guested(B)) **then**
    Stop announcing myself as a beacon

Location Directory Our description so far assumes that the originating node knows the coordinates of the intended destination. Depending on the application, it may be necessary for the originating node to first look up the coordinates by name. We describe a simple mechanism to map node identities to its current coordinates, although this is not the focus of this paper. We propose to use the beacons as a set of storage nodes, by using consistent hashing [14] to provide a mapping $\mathcal{H} : nodeid \mapsto beaconid$, from node ids to the set of beacons. As all nodes know all beacons, any node can independently (and consistently) compute this mapping. The location service consists of two steps: each node k that wishes to be a destination periodically publishes its coordinates to its corresponding beacon $b_k = \mathcal{H}(k)$. Publishing the information entails a self-lookup, which serves as a confirmation. If the coordinates do not change, nodes may choose to refresh their coordinates at a very low rate. Even with changes, we rate limit the update traffic. When a node i wants to route to k, it sends a lookup request to the beacon b_k. Upon receiving a reply, it then routes to the received coordinates. Further communication between the nodes may skip the lookup phase by caching or piggybacking their own location information on the packets they send. We expect that typical data exchanges will be significantly greater in size than these lookup exchanges, so we don't expect that lookup traffic will be a dominant source of sensornet traffic.

This soft-state based approach allows two mechanisms to recover from beacon failures. First, the hashing scheme allows the deterministic choice of backup beacons to replicate the information. The degree of replication depends on the expected failure rate of beacons. Second, the periodic updates will naturally populate a newly elected beacon that replaces a failed beacon. It is important to note that given the redundancy of the coordinate system, even slightly outdated information will lead the routes close to the destination. As we show in our experimental results, both the magnitude and the frequency of the changes to coordinates is small in practice.

4 Simulation Results

To evaluate the BVR algorithm, we use extensive simulations and experiments on testbeds of real sensor motes. To aid the development of BVR and to better understand its behavior and design tradeoffs we start by evaluating BVR using a high-level simulator that abstracts away many of the vagaries of the underlying wireless medium. While clearly not representative of real radios, these simplifications allow us to explore questions of algorithm behavior over a wide range of network sizes, densities, and obstacles that would not be possible on a real testbed.

In practice however, the characteristics of wireless sensor networks impose a number of challenges on actual system development. For example, the `mica2dot` motes have severe resource constraints – just 4KB of RAM, typical packet payloads of 29 bytes *etc.* – and the wireless medium exhibits changing and imperfect connectivity. Hence, our next round of evaluation is at the actual implementation level. We present the implementation and experimental evaluation of our BVR prototype in Sections 5 and 6 respectively and our simulation results in this section.

Our simulator makes several simplifying assumptions. First, it models nodes as having a fixed circular radio range; a node can communicate with all and only those nodes that fall within its range. Second, the simulator

ignores the capacity of, and congestion in, the network. Finally, the simulator ignores packet losses. While these assumptions are clearly unrealistic, they allow the simulator to scale to tens of thousands of nodes. We place nodes uniformly at random in a square planar region, and we vary the total number of beacons r, and the number of routing beacons, k. In all our tests, we compare the results of routing over BVR coordinates to greedy geographic routing over the true positions of the nodes.

Our default simulation scenario uses a 3200 node network with nodes uniformly distributed in an area of 200 $\times$ 200 square units. The radio range is 8 units, and average node degree is 16. Unless otherwise stated, a node's neighbors are those nodes within its one hop radius.

4.1 Metrics

In our evaluation, we consider the following performance metrics:

(Greedy) success rate: The fraction of packets that are delivered to the destination **without requiring flooding**. We stress that the final scoped flooding phase ensures that **all** packets eventually reach their destination. This metric merely measures how often the scoped flooding is not required. Like previous virtual coordinate solutions [24, 25], we report on the success of routing without scoped floods because that provides the most unambiguous evaluation of the quality of node coordinates themselves; *e.g.*, scoped flooding, which will always succeed, does not depend on coordinates. If our results are comparable to those with true positions, then BVR would have overcome the need for geographic information in current proposals. Nonetheless, later in this section, we also present results on a metric we term *transmission stretch* that does explicitly account for the overhead of scoped floods.

Flood scope: The number of hops it takes to reach the destination in those cases when flooding is invoked.

Path stretch: The ratio of the path length of BVR to the path length of greedy routing using true positions.

Node load: The number of packets forwarded per node.

In each test, we're interested in understanding the overhead required to achieve good performance as measured by the above metrics. There are three main forms of overhead in BVR:

Control overhead: This is the total number of flooding messages generated to compute and maintain node coordinates and is directly dependent on r, the total number of beacons in the system. We measure control overhead in terms of the total number of beacons that flood the network. Ideally, we want to achieve high performance with reasonably low r.

Per-packet header overhead: A destination is defined in terms of its $k(\leq r)$ routing beacons. Because the destination position is carried in the header of every packet for routing purposes, k should be reasonably low.

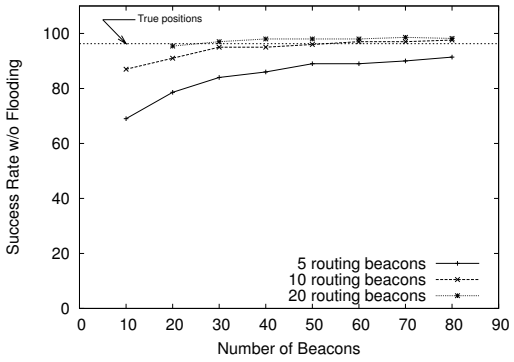


Figure 1: Success rate of routes without flooding in a 3200 node network, for different numbers of total beacons, r, and routing beacons, k.

Routing state: The number of neighbors a node maintains in its routing table.

4.2 Routing Performance vs. Overhead

In this section, we consider the tradeoff between the routing success rate and the flood scope on one hand, and the overhead due to control traffic (r) and per-packet state (k) on the other hand. We use our default simulation scenario and for each of ten repeated experiments, we randomly choose r beacons from the total set of nodes. We vary r from 10 to 80 each time generating $32,000$ routes between randomly selected pairs of nodes.

Figure 1 plots the routing success rate for an increasing total number of beacons (r) at three different values of k, the number of routing beacons ($k = 5$, 10, and 20) As expected, the success rate increases with both the number of total beacons and the number of routing beacons. We draw a number of conclusions from these results. We see that with just $k = 10$ routing beacons we can achieve routing performance comparable to that using true positions. The performance improvement in increasing k to 20 is marginal. Hence, from here on, we limit our tests to using $k = 10$ routing beacons as a good compromise between per-packet overhead and performance. Using $k = 10$, we see that only between 20 to 30 total beacons (r) is sufficient to match the performance of true positions. At less than 1% of the total number of nodes, this is very reasonable flooding overhead. The scope of floods as a function of r decreases from 7 at r =10 to 3 at r =70.

The average path length in these tests was 17.5 hops and the path stretch, i.e., the length of the BVR path over the path length using greedy geographic routing over true positions, is 1.05. In all our tests, we found that the path stretch was always less than 1.1 and hence we don't present path stretch results from here on.

We also compared the distribution of the routing load over nodes using BVR versus greedy geographic routing over true positions and found that for most nodes,

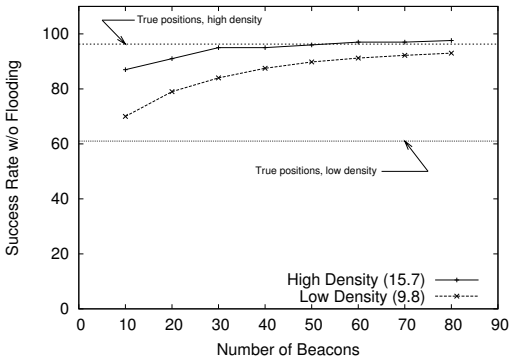


Figure 2: Success rate of routes without flooding, for 3200 node topologies with different densities, for $k = 10$ routing beacons.

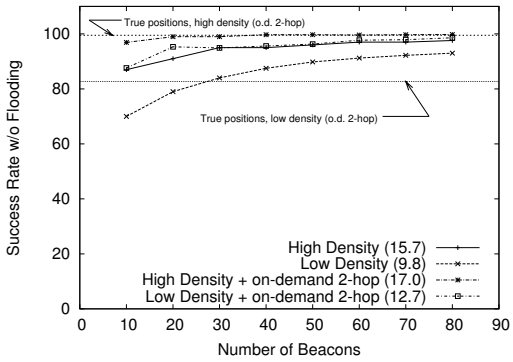


Figure 3: Success rate of routes without flooding, for the same topologies as in figure 2, comparing the on demand acquisition of 2-hop neighborhood information.

the load is virtually identical though BVR does impose slightly higher load on the nodes in the immediate vicinity of beacons. For example, for the above test using $r = 40$ and $k = 10$, the 90th percentile load per node was 48 messages using BVR compared to 37 messages using true positions.

In summary, we see that BVR can roughly match the performance of greedy geographic routing over true positions with a small number of beacons using only its one-hop neighbors.

4.3 The Impact of Node Density

In this section, we consider the impact of the node density on the routing success rate. Figure 2 plots the success rate for the original density of 16 nodes per communication range, and for a lower density of 9.8 nodes per communication range. While at high density the performance of both approaches is comparable, we see that at low densities BVR performs much better than greedy geographic routing with true positions. In particular, while the success rate of the greedy routing is about 61%, the success rate of BVR reaches 80% with 30 beacons, and 90% with 40 beacons. Thus, BVR achieves an almost 30% improvement in the success rate compared to greedy routing with true positions. This is because the node coordinates in BVR are derived from the connectivity information, and not from their geographic positions which may be misleading in the presence of the voids that occur at low densities.

These results reflect the inherent tradeoff between the amount of routing state per node and the success rate of greedy routing. At lower densities, each node has fewer immediate neighbors and hence the performance of greedy routing drops. One possibility to improve the performance of our greedy routing is to have nodes maintain state for nodes beyond their one-hop neighborhood. This however increases the overhead and complexity of maintaining routing state. To retain high success rates without greatly (or needlessly) increasing the routing state per node, we propose the use of *on-demand* two-hop neighbor acquisition. Under this approach, a node starts out using only its immediate (one-hop) neighbors. If it cannot forward a message greedily, it fetches its immediate neighbors' neighbors and adds this two-hop neighbors to its routing table. The intuition behind this approach is that the number of local minima in a graph is far smaller than the total number of nodes. Thus, the on-demand approach to augmenting neighbor state allows only those nodes that require the additional state to incur the overhead of maintaining this state.

Algorithm	avg ngbrs	max ngbrs	% nodes w/ 2hop	avg success
BVR (hi-dens)	15.7	30.7	0	96.1
BVR+2hop (hi-dens)	17.0	67.5	5	99.7
true postns (hi-dens)	15.7	31.7	0	96.3
true postns+2hop (hi)	15.8	48.0	0.7	99.5
BVR (lo-dens)	9.8	22.1	0	89.2
BVR+2hop (lo-dens)	12.7	50.0	15	97.0
true postns (lo-dens)	9.8	22.8	0	61.0
true postns+2hop (lo)	10.7	36.3	6	82.7

Table 3: State requirements using on-demand two hop neighbor acquisition for BVR and true positions at two different network densities. These state requirements are averaged over 10 runs with $k = 10$ and $r = 50$.

To evaluate the effectiveness of using on-demand two-hop neighbor acquisition, we repeat the experiments in Figure 2 using this approach. The results are plotted in Figure 3. Not surprisingly, this approach greatly improves the routing success rate. With only 20 beacons, the success rate of BVR exceeds 99% for the high density network, and 96% for the low density network. Table 3 shows the average and worst case increase in the per-node routing state for both BVR and true positions. Using BVR, at high density, only 5% of nodes fetch their two-hop neighbors while 15% of nodes do so at the lower densities. Thus acquiring two-hop neighbors on demand represents a big win at a fairly low cost.

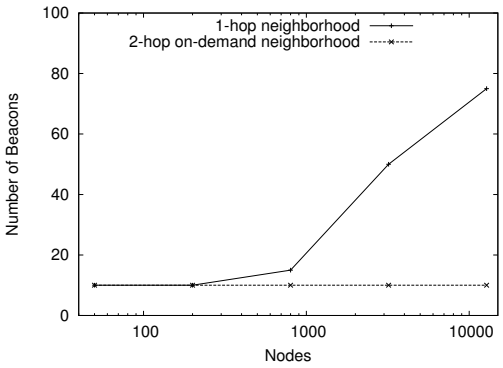


Figure 4: Number of beacons required to achieve less than 5% of scoped floods, with $k = 10$ routing beacons.

4.4 Scaling the Network Size

In this section, we ask the following question: how many beacons are needed to achieve a target success rate as the network size increases? To answer this question, we set the target of the routing success rate at 95%. Figure 4 plots the number of beacons required to achieve this target for both BVR using a one-hop neighborhood, and BVR using on-demand two-hop neighbor acquisition. In both cases the number of routing beacons is 10.

There are two points worth noting. First, the number of beacons for the on-demand two-hop neighborhood remains constant at 10 as the network size increases from 50 to $12,800$ nodes. Second, while the number of beacons in the case of BVR with one-hop neighborhood increases as the network size increases, this number is still very small. When the network is greater than 800 nodes, the number of beacons for the one-hop neighborhood never exceeds 2%.

These results show that the number of beacons required to achieve low flooding rates grows slowly with the size of the system.

4.5 Performance under obstacles

We now study the BVR performance in the presence of obstacles. We model obstacles as horizontal or vertical "walls" with lengths of 10 or 20 units. For comparison, recall that the radio range of a node is 8 units.

Table 4 shows the success rates of BVR routing over a one-hop neighborhood for different numbers of obstacles. For each entry, we also show, in parentheses, the success rate of greedy routing using true positions. Surprisingly, as the number of obstacles and/or their length increases, the decrease in success rate using BVR is *not* significant. In the worst case the success rate drops only from 96% to 91%. For comparison, the success rate of greedy routing with true positions drops from 98% to 43%! Again, this is because the node coordinates in BVR reflect their connectivity instead of their true positions.

Length of Obstacles	Number of Obstacles			
	0	10	20	50
10	0.96 (0.98)	0.96 (0.91)	0.95 (0.87)	0.95 (0.79)
20	0.96 (0.98)	0.95 (0.84)	0.94 (0.70)	0.91 (0.43)

Table 4: Comparing BVR with greedy forwarding over true positions in the presence of obstacles

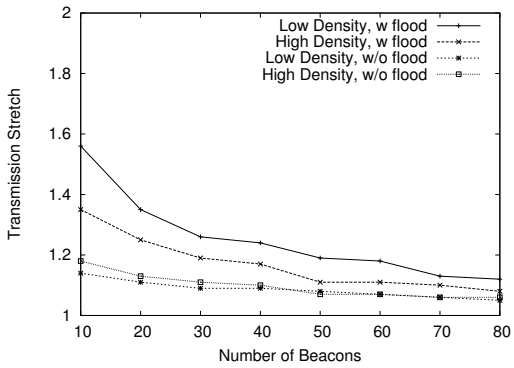


Figure 5: Transmission stretch, average total number of messages sent per route over the the number of messages sent over the shortest path.

4.6 Transmission Stretch

Our results so far evaluated the success of routing without scoped floods. Because scoped flooding incurs higher messaging overhead than unicast forwarding, we now look at a metric we call *transmission stretch*. We measure transmission stretch as the ratio of the *total* (unicast and scoped-flood) number of messages transmitted in routing a packet to that required using the optimal shortest path as computed by Dijkstra's algorithm. Figure 4.5 plots this stretch for BVR routing with and without the use of scoped flood. In the absence of scoped floods we compute stretch only over those routes that do not require floods. We can see that at both low and high density, the transmission stretch improves with increasing number of beacons and rapidly drops to very close to 1. This shows that the use of scoped floods does not incur significant additional overheads. We repeated these tests for network sizes from 50 to 3200 and found that in all cases, the stretch was less than 1.1.

5 BVR Implementation

This section describes our prototype implementation of BVR in TinyOS [11] for the `mica2dot` motes. The resource constraints of the mote hardware and the vagaries of the wireless medium lead to a number of practical difficulties not addressed in our discussion so far. In particular, the following are four key issues that must be addressed in a real implementation:

Link estimation: In a wireless medium, the notion of an individual link is itself ill-defined as the quality of communication varies dramatically across nodes, distance and time. Link estimation is used to characterize a link

as the probability of successful communication rather than a simple binary on/off relation.

Link/neighbor selection: The limited memory in the mote hardware prevents a node from holding state for all its links. Link selection determines the set of neighbors in a node's routing table.

Distance estimation: Recall that our BVR algorithm defines a node's coordinates as its distance in hops to a set of beacons. We describe how we define the hop distance from a node to a beacon when individual links are themselves defined in terms of a quality estimate.

Route selection: This addresses how a node forwards packets in the face of lossy links.

Each of the above is a research problem in itself (see [32, 33] for a detailed exploration of some of these issues); while our implementation makes what we believe are sound choices for each, a comprehensive exploration of the design space for each individual component is beyond the scope of this paper. We describe our solutions to each of the above problems and present the results of our system evaluation in the following section.

Currently, our prototype sets the number of routing beacons equal to the total number of beacons ($k = r$) and does not implement the successive dropping of beacons in computing distances for greedy forwarding (*i.e.*, a node that cannot make greedy progress using all available beacons switches directly to fallback mode). We also do not implement the on-demand neighbor acquisition described in the previous section. If anything, these simplifications can only degrade performance relative to our earlier simulation results.

5.1 Link Estimation and Selection

Estimating the qualities of the links to and from a node is critical to the implementation of BVR as this affects the estimated distance from beacons as well as routing decisions. For example, consider a node that on occasion hears a message directly from a beacon over a low quality link. If, based on these sporadic receptions, the node were to set its distance from the beacon to be one hop then that would have the undesired effect of drawing in traffic over the low quality link.

We implemented a passive link estimator, based on the work by Woo *et al.*[32]. We tag all outgoing packets with a sequence number, such that the receiving nodes can estimate the fraction of packets that are lost from each source. We collect statistics in successive time windows, and the estimation is derived from an exponentially weighted moving average of the quality over time. This estimates the quality of *incoming* links. To accommodate link asymmetry, every node periodically transmits its current list of incoming link qualities. It is aided by the fact that nodes transmit at a minimum rate, making estimation more reliable: in BVR, nodes periodically broadcast "hello" messages used to announce coordinates and maintain the beacon trees. The link estimator is also responsible for detecting "dead" neighbors, and to keep a table with the best quality links.

Because motes have limited memory, a node may not be able to (or may not want to devote the memory needed to) hold state for all the nodes it might hear from. Hence on the one hand we want a node to hold state for its highest quality links but on the other hand the node does not have the memory resources necessary to estimate the quality of all its links. To tackle this problem we use a scheme that guarantees that a node's link will store a set of neighbors with quality above a given low quality replacement threshold L. When a node is first inserted in the link table it is subject to a *probation* period. We set the probation period to be such that the link estimator would have converged to within 10% of the stable quality of the link. An entry in the link table cannot be replaced unless it is past probation and has a link quality below the replacement threshold. In our prototype, we use a link table size n of 18 neighbors, and set L to 20%.

5.2 Distance Estimation

Every node in BVR maintains two key pieces of information: (1) its distance in hops to the root beacons and (2) the positions of the node's immediate neighbors. The only control traffic for maintaining both consists of periodic local neighbor exchanges, in which nodes advertise their distance to each beacon, in the style of distance-vector routing algorithms. A beacon's periodic announcement includes a sequence number that is incremented at every interval. Through periodic neighbor exchanges, nodes build a reverse path tree to every beacon. A node maintains the highest sequence number and a parent along the tree to every beacon. These combined can eliminate count-to-infinity problems, loops, and allows for the detection of dead beacons.

Central to BVR is a node's distance in hops to each beacon. In the presence of lossy links, it is important to avoid using long and unreliable links, which can give the false impression of a low hopcount to the root. To this effect, nodes determine their distance from the beacon by choosing parents that minimize the expected number of transmissions (ETX) to the root [32] along the reverse path. The ETX for one link with forward and reverse transmission success probability p_f and p_r is $\frac{1}{(p_f \times p_r)}$, and the ETX to the root is obtained incrementally by adding the ETX for each link. A node's distance is the number of hops along such path. We use some hysteresis when selecting parents with different hopcounts to increase the stability of the coordinates.

5.3 Route Selection

When selecting the next hop in forwarding a message, our BVR prototype takes into account both the progress in the distance function and the quality of the links. Usu-

Link Estimator	
Size of Table	18
Expiration	5 succ. windows
Replacement Thresh.	20% quality
Reverse Link Info Period	17.5s $\pm$ 50% (jittered)
Update Link Period	30 (fixed)
Exponential Average	smoothing constant 40%
BVRState	
Position Broadcast T	10s $\pm$ 50% (uniform)

Table 5: Parameters used in the experiments on both the *Office-Net* and *Univ-Net* testbeds

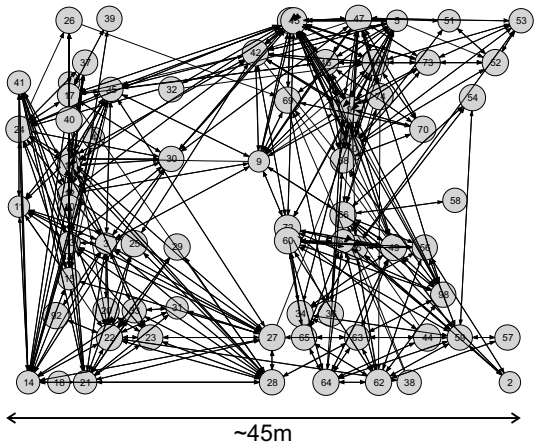


Figure 6: Neighborhood graph as determined by the neighbor tables of motes. Each node is shown with its ID. The positions of the nodes are to scale.

ally, the nodes that make the most progress are also further away, and present poor link quality. We order all links that make some progress towards the destination by *expected progress*: the product of the bidirectional link quality and the actual progress in the distance function. This is analogous to the $PRR \times Distance$ metric from [28], found to be optimal for geographic routing. When sending a message, we use two optimizations for reliability. First, we use link level acknowledgments with up to five retransmissions. Second, if a transmissions fails despite the multiple retries, the node will try the other neighbors in decreasing order of their expected progress. Only when it has exhausted all possible next hop options will the node revert to fallback mode.

Table 5 summarizes our various parameter settings. We selected these based on both our own experience and those reported by Woo *et al.*[32] with the mote radios. We intend to achieve good tradeoff between maintaining freshness of the routing state, and the amount of control traffic generated. A back-of-the-envelope calculation based on our measured channel capacity indicates that our timer settings lead to control traffic of approximately 5% of the channel capacity. Fully understanding the generality of our parameter selection is beyond the scope of this paper and a topic we intend to explore. Nonetheless, because sensornet topologies (unlike most other networks) are dependent on so many deployment specific issues — interference and obstacles in the physical environment, number and layout of nodes, power settings, *etc.* — we do expect deployments to always involve some amount of a priori calibration to guide parameter selection.

6 Prototype Evaluation

This section presents the results of our experiments with the BVR prototype deployed over two testbeds. The first (Office-Net) consists of 42 `mica2dot` motes [10] in an indoor office environment of approximately 20x50m while the second (Univ-Net) is a testbed of about 74 `mica2dot` motes deployed across multiple student offices on a single floor of UC Berkeley's Computer Science building. In both testbeds, motes are connected to an Ethernet backchannel that we use for logging and driving the experiments. These testbeds are of moderate scale, with diameters of five and 7 hops, respectively, and hence do not truly stress BVR's scalability. Nonetheless, these deployments are an invaluable (if not the only!) means by which to test our algorithms under the non-uniform and time-varying radio characteristics that cannot be easily captured in simulation.

On both testbeds, we set parameters as described in the previous section. Our experiments consist of a setup phase of several minutes to allow the link estimations to converge, beacon trees to be constructed and nodes to discover their neighbors' positions. After this setup phase, we issue route commands from a PC to individual motes over the Ethernet backchannel.

In the reminder of this section we evaluate four main aspects of the BVR design:

Link Estimation: We validate that a node indeed selects high quality neighbors. This is important because, as has been reported, the details of link estimation are at once tricky and greatly impact performance. Moreover, because link quality is a function of environment, topology and traffic patterns, we could not just expect behavior identical to previous studies [32, 33].

Routing Performance: We evaluate BVR's success rate on two testbeds under increasing load. We find that the routing success rate is high (over 97%) when the network load is low, and degrades gracefully as the load increases.

Dynamics: We evaluate performance under both node and beacon failures and show that BVR sustains high performance even under high node failure rate.

Coordinate Stability: Because many applications may use node coordinates as addresses, it is important that these coordinates vary little in magnitude and over time. We find that BVR coordinates are quite stable.

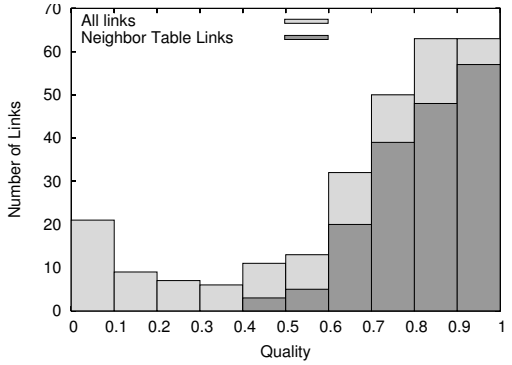


Figure 7: Histograms of the measured link qualities of all links and of the subset of these links chosen that are part of motes' routing tables. Notice how the latter are proportionately better quality.

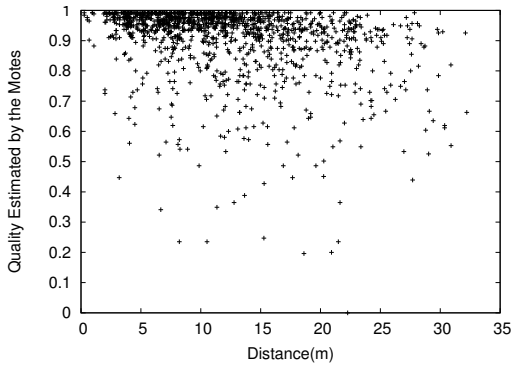


Figure 8: Link quality versus distance. Network connectivity is not always correlated with physical distance.

6.1 Link Estimation

We verify that BVR successfully estimates individual link qualities and selects high quality neighbor links over which to route. Our data also shows little correlation between distance and link quality. Since the evaluation of the link estimator does not require a large multi-hop network, we obtain our results from a subset of twenty three motes in our Office-Net environment. Based on the packets logged at each mote, we record the true quality of every link over which even a single packet was received. Figure 7 compares these measured link qualities to those of the subset of links selected by motes in their routing tables. Note how the fraction of neighbor links selected in each range of quality increases with quality, which confirms that nodes choose links with comparatively good qualities to be part of their coordinate tables.

We also examined the network-wide connectivity in our testbeds. Figure 6 shows a snapshot of the network, drawn to scale, and the connectivity as determined by the neighbor tables at each mote on the 74 node Univ-Net testbed. We see that network connectivity is frequently not congruent with physical distance (*e.g.*, mote pairs 32-30, 32-9, 26-37, 35-27, 54-59). We also note the existence of short but asymmetric links (motes 24-26).

For the same testbed, Figure 8 shows the relation between link quality and physical distance between pairs of nodes that *are* neighbors (as determined by BVR's link and neighbor selection algorithms). While BVR select predominantly high quality neighbors, these are quite frequently not physically close; in fact, a fair number of neighbors are more than halfway across the network. Note that these observations contradict the circular radio assumptions made by typical geographic routing algorithms and lend credibility to the need for connectivity-derived coordinates.

6.2 Routing Performance

In the following experiments, after the setup phase, each mote periodically attempts to route to a random destination mote. We present BVR routing performance in terms of successful packet delivery under increasing routing load. In these tests we do not experiment with beacon selection, and just preconfigure 5 well spread nodes to act as beacons. In the following section we evaluate our implementation of beacon selection using the low-level mote simulator TOSSIM [18].

Figure 9 present results for the Office-Net and Univ-Net testbeds. The graphs show: (1) the overall success rate measured as the fraction of routes that arrived at the target destination, (2) the fraction of routes that required scoped flooding, (3) the fraction of routes that failed due to contention drops where contention drops are packets that were dropped due to a lack of sending buffers along the internal send path in the mote network stack and (4) failures which are routes that failed despite the multiple retries; *i.e.*, the message was repeatedly sent out over the channel but no acknowledgments were received. The graph also plots the aggregate network route request rate over time with the scale on the right hand Y axis. Our tests start (after the setup phase) with a rate of one route request per second for a period of approximately one hour; after this we increase the route request rate every 400 seconds up to a maximum rate of approximately 8 routes/second.

On Office-Net, BVR achieved an average success rate (greedy or flood) of 99.9% until a load of about 8.8 requests/second, at which point we start seeing a small number of contention drops. 1.2% of all route requests in this period resulted in scoped flooding (with an average scope of 2 hops), and less than 0.1% were contention drops. Similarly for Univ-Net, the average success rate was 98.5%. 5.5% of all routes required scoped flooding (with again an average scope of 2 hops), there were no contention drops, and 1.15% of routes failed due to persistent loss. We repeated the above tests with a larger number of beacons and recorded similarly high success

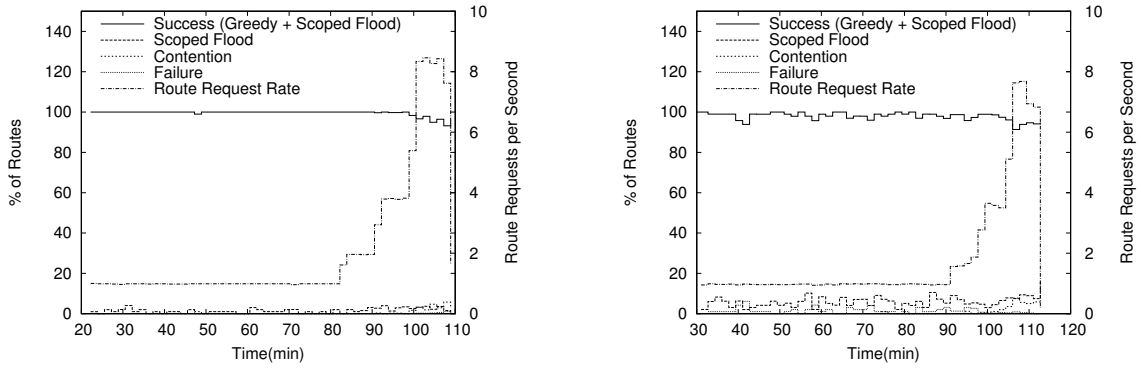


Figure 9: Results of routing tests under increasing routing load, for *Office-Net* on the left and *Univ-Net* on the right. Rate on the right hand Y axis is the aggregate number of route requests issued per second in the network.

rates; due to space considerations, we do not present these here. These experiments indicate that our BVR implementation works correctly in a real deployment, and can sustain a significant workload of routing messages.

6.3 Node Dynamics

Sensor nodes are vulnerable to temporary or permanent failures due to depleted energy resources, or damage from weather conditions. We test BVR's resilience to such failures using both TOSSIM and real testbed experimentation. We show that BVR can recover from both node and beacon failures and sustain good routing performance without incurring high overhead. We first evaluate BVR's robustness to non-beacon node failure using the Office-Net testbed and then evaluate robustness to beacon failure in TOSSIM.

6.3.1 Node Failure

To verify BVR's robustness to node failure in a real deployment, we ran tests on the Office-Net testbed with artificially induced node failures. The setup for this is identical to that in Section 6.2 except that we maintain a query load of one route/second and, after a warming period, repeatedly kill one random (non-beacon) mote every five minutes until all but the beacons nodes have been killed. Note that, once dead, we never resurrect a mote; this failure model is more realistic for sensor networks where the predominant cause of failure is battery exhaustion and not (as on the Internet) node reboots or disconnections [30]. Figure 10 plots the success rate (along with the number of live motes) over time. We see that BVR is extremely resilient to random node failure. The routing success rate remains mostly high until well over 80% of the motes are killed, at which point the success rate drops. Closer examination of the logs revealed that while node failures do lead to occasional dips in success rate, BVR quickly recovers. This behavior stems from the redundancy in the coordinate system, the route selection mechanisms, and the adaptability of the coordinates to the topology changes.

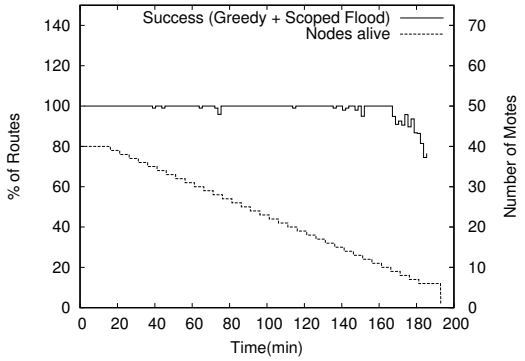


Figure 10: *Office-Net* Success rate (left Y axis) and number of live motes (right Y axis) over time. BVR maintains high success rates under increasing node failure.

6.3.2 Beacon Failure

Section 3 discussed the need to maintain a reasonable number of beacon nodes in the network. Our BVR implementation includes Algorithm 2 which we tested using TOSSIM. Our TOSSIM experiments choose a baseline configuration of 100 motes with 8 beacons and expected node degree of 12. We use TOSSIM's lossy link generator, which is itself based on empirical data and includes lossy and asymmetric connectivity. After a 30 minute setup phase, we initiate a constant rate of one route per second between random node pairs and kill a randomly selected beacon node at increasing rates. Success rates are computed in 100 second time windows under traffic load of 1 route request per second.

Figure 11 shows a typical simulation run. We see that routing performance does not degrade with occasional beacon failures, and tolerates high failure rates reasonably well. This is largely because BVR routes well with even a partial beacon vector set. Residual beacon vectors from recently deceased beacons also serve as hints for packet forwarding. Closer examination of the test log shows that the convergence time of the beacon replenishment is fast, and dependent as expected on the network diameter and frequency of neighbor exchanges. Candi-

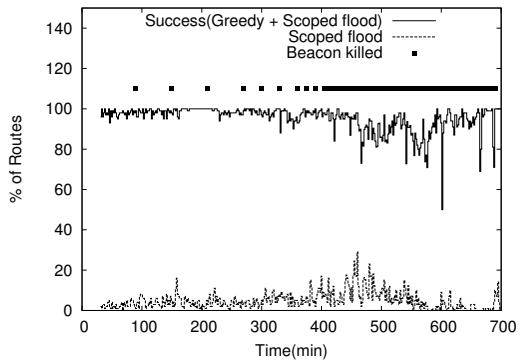


Figure 11: Resilience to beacon failure (TOSSIM): Randomly chosen beacons fail at increasing rates up to a maximum rate of 1/10 per min. Success rates are computed in 100 second windows.

date beacons are efficiently suppressed and hence we observed no significant communication overhead.

6.4 Coordinate Stability

Our results so far have shown that BVR generates good coordinates in that they correctly guide routes towards a target destination. Coordinates stability is important for routing performance, especially for applications that require a location database such as that described in Section 3. Not only may routing to outdated coordinates lead to routing failures, but constant changes can generate heavy update and lookup traffic close to the beacons.

In Figure 12 we look at the number of nodes with changes in coordinates per fixed 100-second intervals for a run in *Office-Net*, with a load of one route per second. This approximates the minimum aggregate update traffic that would be seen in the network if motes were to update at most once every such period, when their coordinates changed. From the graph we see that there are more nodes with changes in the beginning, as the link estimators are stabilizing, and then there is a reduction. The average number of motes with changes per slot is 0.95, and the 95th percentile is 9. We also looked at 10 second intervals, and the average number of motes with changes is 0.13 per interval, which is relatively consistent. In terms of number of changes, during this period of 100 minutes, 90% of the motes had 5 or fewer changes in coordinates. The results we saw in the *Univ-Net* testbed are similar. Figure 13 plots the distribution of the magnitude of individual coordinate changes over all coordinate changes. Magnitude here is simply the change in a node's distance to a beacon; *i.e.*, a change in distance to a beacon from 5 hops to 3 hops would be counted as a magnitude of 2. We see that change, when it occurs, is small: for both testbeds, in this case, at least 80% of the changes were of 2 or less hops. These results suggest that BVR can be used as a stable routing solution that is scalable, robust, and does not unduly load beacons.

7 Conclusions and Future Work

Beacon Vector Routing is a new approach to achieving scalable point-to-point routing in wireless sensornets. Its main advantages are its simplicity, making it easy to implement on resource constrained nodes like motes, and resilience, in that we build no large-scale structures. In fact, the periodic flooding from the beacons means that no matter what failures have occurred, the entire state can be rebuilt after one refresh interval. Our simulation results show that BVR achieves good performance in a wide range of settings, at times significantly exceeding that of geographic routing. Our implementation results suggest that BVR can withstand a testbed environment and thus might be suitable for real deployments.

However, we are at the very early stages of our investigation. We need to better understand how BVR's performance is linked to radio stability, the generality of our parameter selection as well as more rigorous approaches to tuning these parameters. Most importantly however, we have not yet implemented any applications on top of BVR, so we don't yet know if it provides a suitably stable substrate on which to build. All of these items represent future work.

References

[1] BOSE, P., MORIN, P., STOJMENOVIC, I., AND URRUTIA, J. Routing with guaranteed delivery in ad hoc wireless networks. *Wireless Networks 7*, 6 (2001), 609–616.

[2] CAO, Q., AND ABDELZAHER, T. A scalable logical coordinates framework for routing in wireless sensor networks. In *IEEE Real-time Systems Symposium* (December 2004).

[3] COUTO, D. D., AGUAYO, D., CHAMBERS, B., AND MORRIS, R. Performance of multihop wireless networks: Shortest path is not enough. In *Proceedings of HotNets* (Oct. 2002).

[4] DEMIRBAS, M., ARORA, A., AND GOUDA, M. A pursuer-evader game for sensor networks. In *Proceedings of the Sixth Symposium on Self-Stabilizing Systems* (2003), pp. 1–16.

[5] FLOYD, S., JACOBSON, V., MCCANNE, S., LIU, C., AND ZHANG, L. A reliable multicast framework for light-weight sessions and application level framing. In *Proceedings of SIGCOMM* (1995), ACM Press, pp. 342–356.

[6] GANESAN, D., ESTRIN, D., AND HEIDEMANN, J. DIMENSIONS: Why do we need a new data handling architecture for sensor networks? In *Proceedings of the ACM HotNets* (October 2002), ACM, pp. 143–148.

[7] GIROD, L., STATHOPOULOS, T., RAMANATHAN, N., ELSON, J., ESTRIN, D., OSTERWEIL, E., AND SCHOELLHAMMER, T. EmStar: An environment for developing wireless embedded systems software. In *Proceedings of the Second SenSys* (Nov. 2004), ACM Press.

[8] GREENSTEIN, B., ESTRIN, D., GOVINDAN, R., RATNASAMY, S., AND SHENKER, S. DIFS: A distributed index for features in sensor networks. In *Proceedings of First IEEE WSNA* (May 2003).

[9] HAMILTON, M., ALLEN, M., ESTRIN, D., ROTTENBERRY, J., RUNDEL, P., SRIVASTAVA, M., AND SOATTO, S. Extensible sensing system: An advanced network design for microclimate sensing, June 2003.

[10] HILL, J., AND CULLER, D. Mica: a wireless platform for deeply embedded networks. *IEEE Micro 22*, 6 (November 2002), 12–24.

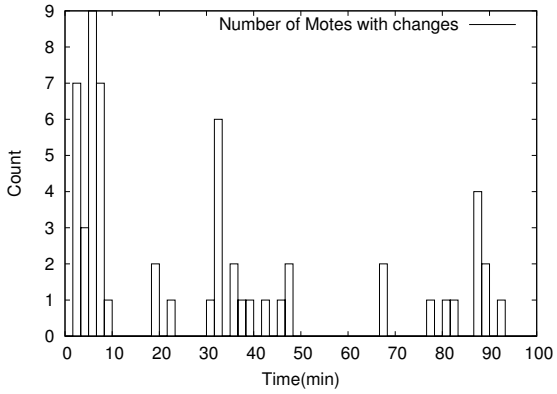


Figure 12: Number of motes with at least one coordinate change per slot of 100 seconds. This approximates the update traffic the location database would see, assuming a minimum update period of 100 seconds per mote.

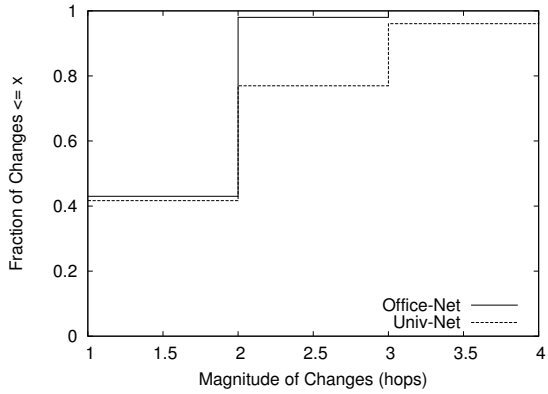


Figure 13: Distribution of the magnitude of individual coordinate changes over all coordinate changes seen across all nodes over test durations of 60 minutes on the Office-Net and Univ-Net testbeds.

[11] HILL, J., SZEWCZYK, R., WOO, A., HOLLAR, S., CULLER, D., AND PISTER, K. System architecture directions for networked sensors. In *Proceedings of the ASPLOS* (2000), ACM Press, pp. 93–104.

[12] INTANAGONWIWAT, C., GOVINDAN, R., AND ESTRIN, D. Directed Diffusion: a scalable and robust communication paradigm for sensor networks. In *Proceedings of the 6th Annual MOBICOM* (2000), ACM Press, pp. 56–67.

[13] JOHNSON, D. B., AND MALTZ, D. A. Dynamic source routing in ad hoc wireless networks. In *Mobile Computing*, Imielinski and Korth, Eds., vol. 353. Kluwer Academic Publishers, 1996.

[14] KARGER, D. R., LEHMAN, E., LEIGHTON, T., LEVINE, M., LEWIN, D., AND PANIGRAHY, R. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. 29th ACM STOC* (May 1997), pp. 654–663.

[15] KARP, B., AND KUNG, H. T. GPSR: Greedy perimeter stateless routing for wireless networks. In *Proceedings of the 6th annual MOBICOM* (2000), ACM Press, pp. 243–254.

[16] KUHN, F., WATTENHOFER, R., ZHANG, Y., , AND ZOLLINGER, A. Geometric ad-hoc routing: Of theory and practice. In *22nd ACM PODC* (2003).

[17] KUMAR, S., ALAETTINGLU, C., AND ESTRIN, D. Scalable object-tracking through unattended techniques (scout). In *Proceedings of the 2000 International Conference on Network Protocols* (2000), IEEE Computer Society, p. 253.

[18] LEVIS, P., LEE, N., WELSH, M., AND CULLER, D. Tossim: accurate and scalable simulation of entire TinyOS applications. In *Proceedings of the First SenSys* (2003), ACM Press, pp. 126–137.

[19] LEVIS, P., MADDEN, S., GAY, D., POLASTRE, J., SZEWCZYK, R., WOO, A., BREWER, E., AND CULLER, D. The emergence of networking abstractions and techniques in tinyos. In *Proceedings of the First USENIX/ACM NSDI* (March 2004).

[20] LI, X., KIM, Y. J., GOVINDAN, R., AND HONG, W. Multidimensional range queries in sensor networks. In *Proceedings of the First SenSys* (2003), ACM Press, pp. 63–75.

[21] MADDEN, S. *The design and evaluation of a query processing architecture for sensor networks.* PhD thesis, UC Berkeley, 2003.

[22] MADDEN, S., FRANKLIN, M., HELLERSTEIN, J., AND HONG, W. TAG: a tiny aggregation service for ad hoc sensor networks. In *OSDI* (2002).

[23] MAINWARING, A., POLASTRE, J., SZEWCZYK, R., CULLER, D., AND ANDERSON, J. Wireless sensor networks for habitat monitoring. In *Proceedings of ACM WSNA* (Sept. 2002).

[24] NEWSOME, J., AND SONG, D. Gem: Graph embedding for routing and data-centric storage in sensor networks without geographic information. In *Proceedings of the First SenSys* (2003), ACM Press, pp. 76–88.

[25] RAO, A., RATNASAMY, S., PAPADIMITRIOU, C., SHENKER, S., AND STOICA, I. Geographic routing without location information. In *Proceedings of the 9th Annual MOBICOM* (2003), ACM Press, pp. 96–108.

[26] ROYER, E. M., AND TOH, C.-K. A review of current routing protocols for ad-hoc mobile wireless networks. *IEEE Personal Communications 6* (April 1999), 46–55.

[27] SAVVIDES, A., HAN, C.-C., AND SRIVASTAVA, M. B. Dynamic fine-grained localization in ad-hoc networks of sensors. In *Mobile Computing and Networking* (2001), pp. 166–179.

[28] SEADA, K., ZUNIGA, M., HELMY, A., AND KRISHNAMACHARI, B. Energy-efficient forwarding strategies for geographic routing in lossy wireless sensor networks. In *Proceedings of the 2nd SenSys* (2004), ACM Press, pp. 108–121.

[29] SHENKER, S., RATNASAMY, S., KARP, B., GOVINDAN, R., AND ESTRIN, D. Data-centric storage in sensornets. *SIGCOMM Comput. Commun. Rev. 33*, 1 (2003), 137–142.

[30] SZEWCZYK, R., POLASTRE, J., MAINWARING, A., ANDERSON, J., AND CULLER, D. An analysis of a large scale habitat monitoring application. In *Proceedings of the Second SenSys* (2004), ACM Press.

[31] TSUCHIYA, P. F. The Landmark Hierarchy: a new hierarchy for routing in very large networks. In *ACM SIGCOMM* (1988), ACM Press, pp. 35–42.

[32] WOO, A., TONG, T., AND CULLER, D. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of the First SenSys* (2003), ACM Press, pp. 14–27.

[33] ZHAO, J., AND GOVINDAN, R. Understanding packet delivery performance in dense wireless sensor networks. In *Proceedings of the First SenSys* (2003), ACM Press, pp. 1–13.

[34] ZHAO, J., GOVINDAN, R., AND ESTRIN, D. Computing aggregates for monitoring wireless sensor networks. In *Proceedings of the IEEE SNPA* (May 2003).

Active Sensor Networks

Philip Levis[†], David Gay[‡], and David Culler[†]
{pal,culler}@cs.berkeley.edu, david.e.gay@intel.com

[†]EECS Department
University of California, Berkeley
Berkeley, CA 94720

[‡]Intel Research Berkeley
2150 Shattuck Avenue
Berkeley, CA 94703

ABSTRACT

We propose using application specific virtual machines (ASVMs) to reprogram deployed wireless sensor networks. ASVMs provide a way for a user to define an application-specific boundary between virtual code and the VM engine. This allows programs to be very concise (tens to hundreds of bytes), making program installation fast and inexpensive. Additionally, concise programs interpret few instructions, imposing very little interpretation overhead. We evaluate ASVMs against current proposals for network programming runtimes and show that ASVMs are more energy efficient by as much as 20%. We also evaluate ASVMs against hand built TinyOS applications and show that while interpretation imposes a significant execution overhead, the low duty cycles of realistic applications make the actual cost effectively unmeasurable.

1. INTRODUCTION

Wireless sensor networks have limited resources and tight energy budgets. These constraints make in-network processing a prerequisite for scalable and long-lived applications. However, as sensor networks are embedded in uncontrolled environments, a user often does not know exactly what the sensor data will look like, and so must be able to reprogram sensor network nodes after deployment. Proposals for domain specific languages — still an area of open investigation [5, 7, 19, 21, 23, 28] — present possible programming models for writing these programs. TinySQL queries, for example, declare how nodes should aggregate data as it flows to the root of a collection tree.

This wide range of programming abstractions has led to a similarly wide range of supporting runtimes, ranging from in-network query processors [19] to native thread libraries [28] to on-node script interpreters [5]. However, each is a vertically integrated solution, making them all mutually incompatible with each other. Additionally, they all make implementation assumptions or simplifications that lead to unnecessary inefficiencies.

Rather than propose a new programming approach to in-network processing, in this paper we propose an architecture for *implementing* a programming model's underlying runtime. We extend our prior work on the Maté virtual machine (a tiny bytecode interpreter) [15], generalizing its simple VM into an architecture for building application specific virtual machines (ASVMs). Our experiences showed that Maté's harsh limitations and complex instruction set precluded supporting higher level programming. By carefully relaxing some of these restrictions and allowing a user to customize both the instruction set and execution triggering events, ASVMs can support dynamically reprogramming for a wide range of application domains.

Introducing lightweight scripting to a network makes it easy to process data at, or very close to, its source. This processing can improve network lifetime by reducing network traffic, and can improve scalability by performing local operations locally. Similar approaches have appeared before in other domains. Active disks proposed pushing computation close to storage as a way to deal with bandwidth limitations [1], active networks argued for introducing in-network processing to the Internet to aid the deployment of new network protocols [27], and active services suggested processing at IP end points [2]. Following this nomenclature, we name the process of introducing dynamic computation into a sensor network *active sensor networking*. Of the prior efforts, active networking has the most similarity, but the differing goals and constraints of the Internet and sensor networks lead to very different solutions. We defer a detailed comparison of the two until Section 6.

Pushing the boundary toward higher level operations allows application level programs to achieve very high code density, which reduces RAM requirements, interpretation overhead, and propagation cost. However, a higher boundary can sacrifice flexibility: in the most extreme case, an ASVM has a single bytecode, "run program." Rather than answer the question of where the boundary should lie — a question whose answer depends on the application domain — ASVMs provide flexibility to an application developer, who can pick the right level of abstraction based on the particulars of a deployment.

Generally, however, we have found that very dense bytecodes do not sacrifice flexibility, because ASVMs are customized for the domain of interest. RegionsVM, presented in Section 4, is an ASVM designed for vehicle tracking with extensions for regions based operations [28]; typical vehicle tracking programs are on the

order of seventy bytes long, 1/200th the size of the originally proposed regions implementation. A second ASVM we have built, QueryVM, supports an SQL interface to a sensor network at 5–20% lower energy usage than the TinyDB system [19], and also allows adding new aggregation functions dynamically.

This paper has two contributions. First, it shows a way to introduce a flexible boundary between dynamic and static sensor network code, enabling active sensor networking at a lower cost than prior approaches while simultaneously gaining improvements in safety and expressiveness. Second, this paper presents solutions to several technical challenges faced by such an approach, which include extensible type support, concurrency control, and code propagation. Together, we believe these results suggest a general methodology for designing and implementing runtimes for in-network processing.

In the next section, we describe background information relevant to this work, including mote network resource constraints, operating system structure, and the first version of Maté. From these observations, we derive three ways in which Maté is insufficient, establishing them as requirements for in-network processing runtimes to be effective. In Section 3, we present ASVMs, outlining their structure and decomposition. In Section 4 we evaluate ASVMs with a series of microbenchmarks, and compare ASVM-based regions and TinySQL to their original implementations. We survey related work in Section 5, discuss the implications of these results in Section 6, and conclude in Section 7.

2. BACKGROUND

ASVMs run on the TinyOS operating system, whose programming model affects their structure and implementation. The general operating model of TinyOS networks (very low duty cycle) and network energy constraints lead to both very limited node resources and underutilization of those resources. Maté is a prior, monolithic VM we developed for one particular application domain. From these observations, we derive a set of technical challenges for a runtime system to support active sensor networking.

2.1 TinyOS/nesC

TinyOS is a popular sensor network operating system designed for mote platforms. The nesC language [6], used to implement TinyOS and its applications, provides two basic abstractions: component based programming and low overhead, event driven concurrency.

Components are the units of program composition. A component has a set of *interfaces* it *uses*, and a set of interfaces it *provides*. A programmer builds an application by connecting interface users to providers. An interface can be *parameterized*. A component with a parameterized interface has many copies of the interface, distinguished by a parameter value (essentially, an array of the interface). Parameterized interfaces support runtime dispatch between a set of components. For example, the ASVM scheduler uses a parameterized interface to issue instructions: each instruction is an instance of the interface, and the scheduler dispatches on the opcode value.

TinyOS's event-driven concurrency model does not allow blocking operations. Calls to long-lasting operations, such as sending a packet, are typically split-phase: the call to begin the operation returns immediately, and the called component signals an event to the caller on completion. nesC programming binds these callbacks statically at compile time through nesC interfaces (instead of, e.g., using function pointers passed at run-time).

2.2 Mote Networks

As motes need to be able to operate unattended for months to years, robustness and energy efficiency are their dominant system requirements. Hardware resources are very limited, to minimize energy consumption. Current TinyOS motes have a 4–8MHz microcontroller, 4–10kB of data RAM, 60–128kB of program flash memory, and a radio with application-level data transmission rates of 1–20kB/s.

Energy limitations force long term deployments to operate at a very low utilization. Even though a mote has very limited resources, in many application domains some of those resources are barely used. For example, in the 2003 Great Duck Island deployment [26], motes woke up from deep sleep every five or twenty minutes, warmed up sensors for a second, and transmitted a single data packet with readings. During the warm-up second, the CPU was essentially idle. All in all, motes were awake 0.1% of the time, and when awake used 2% of their CPU cycles and network bandwidth. Although a mote usually does very little when awake, there can also be flurries of activity, as nodes receive messages to forward from routing children or link estimation updates from neighbors.

2.3 Maté v1.0

We designed and implemented the first version of Maté in 2002, based on TinyOS 0.6 (pre-nesC) [15]. At that time, the dominant hardware platform was the rene2 (the mica was just emerging), which had 1kB of RAM, 16kB of program memory and a 10kbps software controlled radio. Maté has a predefined set of three events it executes in response to. RAM constraints limited the code for a particular event handler to 24 bytes long (a single packet). In order to support network protocol implementations in this tiny amount of space, the VM had a complex instruction set open to inventive assembly programming but problematic as a compilation target.

2.4 Requirements

Maté's hard virtual/native boundary prevents it from being able to support a range of programming models. In particular, it fails to meet three requirements:

Flexibility: The Maté VM has very concise programs, but is designed for a single application domain. To provide support for in-network processing, a runtime must be flexible enough to be customized to a wide range of application domains. Supporting a range of application domains requires two forms of customization: the execution primitives of the VM (its instruction set), and the set of events it executes in response to. For example, data collection networks need to execute in response to a request to forward a packet up a collection tree (for suppression/aggregation), while a vehicle tracking network needs to execute in response to receiving a local broadcast from a neighbor.

Concurrency: By introducing a lightweight threading model on top of event-driven TinyOS, Maté provides a greatly simplified programming interface while enabling fine-grained parallelism. Limited resources and a constrained application domain allowed Maté to address the corresponding synchronization and atomicity issues by only having a single shared variable. This restriction is not suitable for all VMs. However, forcing explicit synchronization primitives into programs increases their length and places the onus of correctness on the programmer, who may not be an expert on concurrency. Instead, the runtime should manage concurrency automatically, running handlers race-free and deadlock-free while allowing safe parallelism.

Propagation: In Maté, handlers can explicitly forward code with the `forw` and `forwo` instructions. As every handler could fit in a single packet, these instructions were just a simple broadcast. One one hand, explicit code forwarding allows user programs to control their propagation, introducing additional flexibility; on the other, it requires every program to include propagation algorithms, which can be hard to tune and easy to write incorrectly. Maté's propagation data showed how a naive propagation policy can easily saturate a network, rendering it unresponsive and wasting energy. As not all programming models can fit their programs in a single packet, a runtime needs to be able to handle larger data images (e.g., between 20 and 512 bytes), and should provide an efficient but rapid propagation service.

Our prior work on the Trickle [17] algorithm deals with one part of the propagation requirement, proposing a control algorithm to quickly yet efficiently detect when code updates are needed. The propagation results in that work assumed code could fit in a single packet and just broadcasts updates three times. This leaves the need for a protocol to send code updates for larger programs: we present our solution to this problem in Section 3.4.

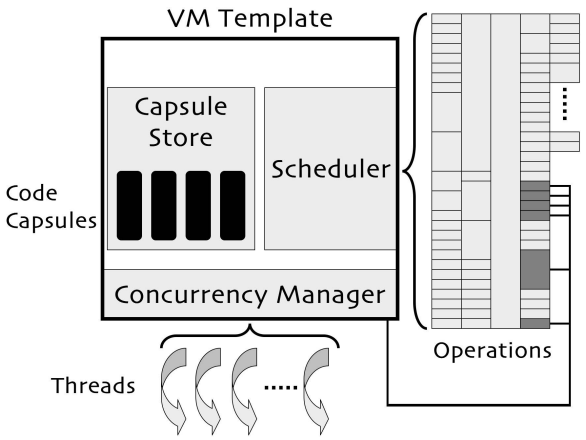


Figure 1: The ASVM architecture.

To provide useful systems support for a wide range of programming models, a runtime must meet these three requirements without imposing a large energy burden. Flexibility requires a way to build customized VMs — a VM generator — so a VM can be designed for an application domain. The next section describes our application specific virtual machine (ASVM) architecture, designed to take this next step.

3. DESIGN

Figure 1 shows the ASVM functional decomposition. ASVMs have three major abstractions: *handlers*, *operations*, and *capsules*. Handlers are code routines that run in response to system events, operations are the units of execution functionality, and capsules are the units of code propagation. ASVMs have a threaded execution model and a stack-based architecture.

The components of an ASVM can be separated into two classes: the *template*, which every ASVM includes, and *extensions*, the application-specific components that define a particular ASVM. The template includes a scheduler, concurrency manager, and capsule store. The scheduler executes runnable threads in a FIFO round-robin fashion. The concurrency manager controls what threads are runnable, ensuring race-free and deadlock-free handler execution. The capsule store manages code storage and loading, propagating code capsules and notifying the ASVM when new code arrives.

Building an ASVM involves connecting handlers and operations to the template. Each handler is for a specific system event, such as receiving a packet. When that event occurs, the handler triggers a thread to run its code. Generally, there is a one-to-one mapping between handlers and threads, but the architecture does not require this to be the case. The concurrency manager uses

```
interface Bytecode {
/* The instr parameter is necessary for primitives
   with embedded operands (the operand is instr
   - opcode). Context is the executing thread. */

  command result_t execute(uint8_t instr,
                           MateContext* context);
  command uint8_t byteLength();
}
```

Figure 2: The nesC Bytecode interface, which all operations provide.

a conservative, flow insensitive and context insensitive program analysis to provide its guarantees.

The set of operations an ASVM supports defines its instruction set. Just as in Maté, instructions that encapsulate split-phase TinyOS abstractions provide a blocking interface, suspending the executing thread until the split-phase call completes. Operations are defined by the Bytecode nesC interface, shown in Figure 2, which has two commands: `execute` and `byteLength`. The former is how a thread issues instructions, while the latter lets the scheduler correctly control the program counter. Currently, ASVMs support three languages: TinyScript, motlle, and TinySQL, which we present in Sections 4.3 and 4.4.

There are two kinds of operations: *primitives*, which are language specific, and *functions*, which are language independent. The distinction between primitives and functions is an important part of providing flexibility. An ASVM supports a particular language by including the primitives it compiles to, while a user tailors an ASVM to a particular application domain by including appropriate functions and handlers. For functions to work in any ASVM, and correspondingly any language, ASVMs need a minimal common data model. Additionally, some functions (e.g., communication) should be able to support language specific data types without knowing what they are. These issues are discussed in Section 3.1. In contrast, primitives can assume the presence of data types and can have embedded operands. For example, conditional jumps and pushing a constant onto the operand stack are primitives, while sending a packet is a function.

The rest of this section presents the ASVM data model and the three core components of the template (scheduler, concurrency manager, and capsule store). It concludes with an example of building an ASVM for region programming.

3.1 Data Model

An ASVM has a stack architecture. Each thread has an operand stack for passing data between operations. The template does not provide any program data storage beyond the operand stack, as such facilities are language specific, and correspondingly defined by primitives. The architecture defines a minimal set of standard simple operand types as 16-bit values (integers and sensor readings); this is enough for defining many useful language-independent functions.

Operation	Width	Name	Operand Bits	Description
rand	1	rand	0	Random 16-bit number
pushc6	1	pushc	6	Push a constant on stack
2jumps10	2	jumps	10	Conditional jump

Table 1: Three example operations: rand is a function, pushc6 and 2jumps10 are primitives.

However, to be useful, communication functions need more elaborate types. For example, the `bcast` function, which sends a local broadcast packet, needs to be able to send whatever data structures its calling language provides. The function takes a single parameter, the item to broadcast, which a program pushes onto the operand stack before invoking it. To support these kinds of functions, languages must provide serialization support for their data types. This allows `bcast`'s implementation to pop an operand off the stack and send a serialized representation with the underlying TinyOS `sendMsg()` command. When another ASVM receives the packet, it converts the serialized network representation back into a VM representation.

3.2 Scheduler: Execution

The core of an ASVM is a simple FIFO thread scheduler. This scheduler maintains a run queue, and interleaves execution at a very fine granularity (a few operations). The scheduler executes a thread by fetching its next bytecode from the capsule store and dispatching to the corresponding operation component through a nesC parameterized interface. The parameter is an 8-bit unsigned integer: an ASVM can support up to 256 distinct operations at its top-level dispatch. As the scheduler issues instructions through nesC interfaces, their selection and implementation is completely independent of the template and the top level instruction decode overhead is constant.

Primitives can have embedded operands, which can cause them to take up additional opcode values. For example, the `pushc6` primitive, which pushes a 6-bit constant onto the operand stack, has six bits of embedded operand and uses 64 opcode slots. Some primitives, such as jump instructions, need embedded operands longer than 8 bits. Primitives can therefore be more than one byte wide.

When the ASVM toolchain generates an instruction set, it has to know how many bits of embedded operand an operation has, if any. Similarly, when the toolchain's assembler transforms compiled assembly programs into ASVM-specific opcodes, it has to know how wide instructions are. All operations follow this naming convention:

`[width]<name>[operand]`

Width and operand are both numbers, while name is a string. Width denotes how many bytes wide the operation is (this corresponds to the `byteWidth` command of the Bytecode interface), while operand is how many bits of embedded operand the operation has. If an operation does not have a width field, it defaults to 1; if it does not have an operand field, it defaults to zero. Table 1 shows three example operations.

Beyond language independence, the function/primitive distinction also determines which operations can be called indirectly. Some languages have first class functions or function pointers: a program must be able to invoke them dynamically, rather than just statically through an instruction. To support this functionality, the scheduler maintains a function identifier to function mapping. This allows functions to be invoked through identifiers that have been stored in variables. For example, when the motlle language calls a function, it pushes a function ID onto the operand stack and issues the `mcall` instruction, which creates a call stack frame and invokes the function through this level of indirection.

3.3 Concurrency Manager: Parallelism

Handlers run in response to system events, and the scheduler allows multiple handler threads to run concurrently. In languages with shared variables, this can easily lead to race conditions, which are very hard to diagnose and detect in embedded devices. The common solution to provide race free execution is explicit synchronization written by the programmer. However, explicit synchronization operations increase program size and complexity: the former costs energy and RAM, the latter increases the chances that, after a month of deployment, a scientist discovers that all of the collected data is invalid and cannot be trusted. One common case where ASVMs need parallelism is network traffic, due the limited RAM available for queuing. One handler blocking on a message send should not prevent handling message receptions, as their presence on the shared wireless channel might be the reason for the delay.

The concurrency manager of the ASVM template supports race free execution through implicit synchronization based on a handler's operations. An operation component can register with the concurrency manager (at compile time, through nesC wiring) to note that it accesses a shared resource. When the ASVM installs a new capsule, the concurrency manager runs a conservative, context-insensitive and flow-insensitive analysis to determine which shared resources each handler accesses. This registration with the concurrency manager is entirely optional. If a language prefers explicit synchronization, then its operations can not declare shared resources, and the concurrency manager will not limit parallelism.

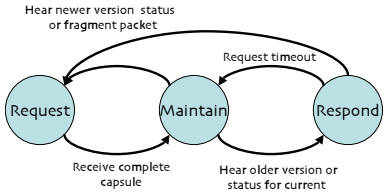


Figure 3: ASVM capsule propagation state machine.

When a handler event occurs, the handler's implementation submits a run request to the concurrency manager. The concurrency manager only allows a handler to run if it can exclusively access all of the shared resources it needs. The concurrency manager enforces two-phase locking: when it starts executing, the handler's thread has to hold all of the resources it may need, but can release them as it executes. When a handler completes (executes the `halt` operation), its thread releases all of the held resources. Releases during execution are explicit operations within a program. If a thread accesses a resource it does not hold (e.g., it incorrectly released it) the VM triggers an error. Two phase locking precludes deadlocks, so handlers run both race free and deadlock free.

When new code arrives, a handler may have variables in an inconsistent state. Waiting for every handler to complete before installing a new capsule is not feasible, as the update may, for example, be to fix an infinite loop bug. Therefore, when new code arrives, the concurrency manager reboots the ASVM, resetting all variables.

The implicit assumption in this synchronization model is that handlers are short running routines that do not hold onto resources for very long. As sensor network nodes typically have very low utilization, this is generally the case. However, a handler that uses an infinite loop with a call to `sleep()`, for example, can block all other handlers indefinitely. Programming models and languages that prefer this approach can use explicit synchronization, as described above.

3.4 Capsule Store: Propagation

Field experience with current sensor networks has shown that requiring physical contact can be a cause of many node failures [25]; network programming is critical. Thus, ASVMs must provide reliable code propagation. As mentioned earlier (Section 2.4), Maté's explicit code forwarding mechanism is problematic. As demonstrated in our work on Trickle [17], the cost of propagation is very low compared to the accompanying control traffic, so selective dissemination enables few energy gains. The ASVM template's capsule store therefore follows a policy of propagating new code to every node. Rather than selective propagation, ASVMs use a policy of selective execution: everyone has the code, but only some nodes execute it.

Trickle is a suppression algorithm for detecting when nodes need code updates. The algorithm dynamically scales its suppression intervals to rapidly detect inconsistencies but sends few packets when the network is consistent. Trickle does not define how code itself propagates, as the protocol greatly depends on the size of the data item. Deluge, for example, transfers entire TinyOS binaries, and so uses a cluster formation algorithm to quickly propagate large amounts of data [11]. In the Maté virtual machine, with its single packet programs, propagation was just a simple local broadcast.

ASVM programs are between these two extremes. As they are on the order of one to twenty packets long, Deluge is too heavy-weight a protocol, and simple broadcasts are not sufficient. To propagate code, the ASVM capsule store maintains three network *trickles* (independent instances of the Trickle algorithm):

- **Version packets**, which contain the 32-bit version numbers of all installed capsules,
- **Capsule status packets**, which describe what fragments a mote needs (essentially, a bitmask), and
- **Capsule fragments**, which are pieces of a capsule.

An ASVM can be in one of three states: maintain (exchanging version packets), request (sending capsule status packets), or respond (sending fragments). Nodes start in the maintain state. Figure 3 shows the state transition diagram. The transitions prefer requesting over responding; a node will defer forwarding capsules until it thinks it is completely up to date.

Each type of packet (version, capsule status, and capsule fragment) is a separate network trickle. For example, a capsule fragment transmission can suppress other fragment transmissions, but not version packets. This allows meta-data and data exchanges to occur concurrently. Trickling fragments means that code propagates in a slow and controlled fashion, instead of as quickly as possible. This is unlikely to significantly disrupt any existing traffic, and prevents network overload. We show in Section 4.2 that because ASVM programs are small they propagate rapidly across large multi-hop networks.

3.5 Building an ASVM

Building an ASVM and scripting environment requires specifying three things: a language, functions, and handlers. Figure 4 shows the description file for RegionsVM, an ASVM that supports programming with regions [28]. We evaluate RegionsVM versus a native regions implementation in Section 4.2. The final HANDLER line specifies that this ASVM executes in response to only one event, when the ASVM boots (or reboots). ASVMs can include multiple handlers, which usually leads to multiple threads; RegionsVM, following the regions programming model of a single execution context, only includes one, which runs when the VM reboots. From this file, the toolchain generates TinyOS source code implementing the ASVM, and the Java classes its assembler uses to map assembly to ASVM opcodes.

```
<VM NAME="KNearRegions" DIR="apps/RegionsVM">

<LANGUAGE NAME="tinyscript">

<FUNCTION NAME="send">
<FUNCTION NAME="mag">
<FUNCTION NAME="cast">
<FUNCTION NAME="id">
<FUNCTION NAME="sleep">
<FUNCTION NAME="KNearCreate">
<FUNCTION NAME="KNearGetVar">
<FUNCTION NAME="KNearPutVar">
<FUNCTION NAME="KNearReduceAdd">
<FUNCTION NAME="KNearReduceMaxID">
<FUNCTION NAME="locx">
<FUNCTION NAME="locy">

<HANDLER NAME="Boot">
```

Figure 4: Minimal description file for the RegionsVM. Figure 7 contains scripts for this ASVM.

3.6 Active Sensor Networking

In order to support in-network processing, ASVMs must be capable of operating on top of a range of single-hop and multi-hop protocols. Currently, the ASVM libraries support four concrete networking abstractions through functions and handlers: single hop broadcasts, any-to-one routing, aggregated collection routing, and abstract regions [28]. Based on our experiences writing library ASVM components for these protocols — 80 to 180 nesC statements — including additional ones as stable implementations emerge should be simple and painless.

4. EVALUATION

We evaluate whether ASVMs efficiently satisfy the requirements presented in Section 2: concurrency, propagation, and flexibility. We first evaluate the three requirements through examples and microbenchmarks, then evaluate overall application level efficiency in comparison to alternative approaches.

In our microbenchmarks, cycle counts are from a mica node, which has a 4MHz 8-bit microcontroller, the ATMega103L; some members of the mica family have a similar MCU at a faster clock rate (8MHz). Words are 16 bits and a memory access takes two cycles: as it is an 8-bit architecture, moving a word (or pointer) between memory and registers takes 4 clock cycles.

4.1 Concurrency

We measured the overhead of ASVM concurrency control, using the cycle counter of a mica mote. Table 2 summarizes the results. All values are averaged over 50

Operation	Cycles	Time (μs)
Lock	32	8
Unlock	39	10
Run	1077	269
Analysis	15158	3790

Table 2: Synchronization Overhead. Lock and unlock are acquiring or releasing a shared resource. Run is moving a thread to the run queue, obtaining all of its resources. Analysis is a full handler analysis.

	Mean	Std. Dev.	Worst
Mote Retasking	20.8s	7.4s	85.8s
Network Retasking	39.5s	10.4s	85.8s
Vector Packets Sent	1.0	1.1	13
Status Packets Sent	3.2	2.4	19
Fragment Packets Sent	3.0	2.5	22
Total Packets Sent	7.3	3.6	30

Table 3: Propagation data. Mote retasking is across all motes in all experiments. Network retasking is the retasking times in all the experiments, based on the time for the last mote to reprogram. The Packets Sent are all on a per-mote basis.

samples. These measurements were on an ASVM with 24 shared resources and a 128 byte handler. Locking and unlocking resources take on the order of a few microseconds, while a full program analysis for shared resource usage takes under a millisecond, approximately the energy cost of transmitting four bits.

These operations enable the concurrency manager to provide race-free and deadlock-free handler parallelism at a very low cost. By using implicit concurrency management, an ASVM can prevent many race condition bugs while keeping programs short and simple.

4.2 Propagation

To evaluate code propagation, we deployed an ASVM on a 71 mote testbed in Soda Hall on the UC Berkeley campus. The network topology was approximately eight hops across, with four hops being the average node distance. We used the standard ASVM propagation parameters.[1] We injected a one hundred byte (four fragment) handler into a single node over a wired link. We repeated the experiment fifty times, resetting the nodes after each test to restore the trickle timers to their stable values (maximums).

[1]Status and version packets have a τ range of one second to twenty minutes, and a redundancy constant of 2. Fragments use Trickle for suppression, but operate with a fixed window size of one second repeating twice, and have a redundancy constant of 3. The request timeout was five seconds.

	Native	RegionsVM
Code (Flash)	19kB	39kB
Data (RAM)	2775B	3017B
Transmitted Program	19kB	71B

Table 4: Space utilization of native and RegionsVM regions implementations (bytes).

```
buffer packet;

bclear(packet);
packet[0] = light();

send(packet);
```

(a) TinyScript

```
bpush3 3
bclear
light
pushc6 0
bpush3 3
bwrite
bpush3 3
send
```

(b) ASVM Bytecodes

Figure 5: TinyScript function invocation on a simple sense and send loop. The operand stack passes parameters to functions. In this example, the scripting environment has mapped the variable "packet" to buffer three. The compiled program is nine bytes long.

Table 3 summarizes the results. On the average, the network reprogrammed in forty seconds, and the worst case was eighty-five seconds. To achieve this rate, each node, on the average, transmitted seven packets, a total of five hundred transmissions for a seventy node network. The worst case node transmitted thirty packets. Checking the traces, we found this mote was the last one to reprogram in a particular experiment, and seems to have suffered from bad connectivity or inopportune suppressions. It transmitted eleven version vectors and nineteen status packets, repeatedly telling nodes around it that it needed new code, but not receiving it. With the parameters we used, a node in a stable network sends at most three packets per hour.

To evaluate the effect ASVM code conciseness has on propagation efficiency, we compare the retasking cost of the native implementation proposed for regions versus the cost of retasking a system with RegionsVM. In the regions proposal, users write short nesC programs for a single, synchronous "fiber" that compile to a TinyOS binary. Reprogramming the network involves propagating this binary into the network. As regions compiles to native TinyOS code, it has all of the safety issues of not having a protection boundary.

Abstract regions is designed to run in TOSSIM, a simulator for TinyOS [16]. Several assumptions in its protocols — such as available bandwidth — prevent it from running on motes, and therefore precluded us from measuring energy costs empirically. However, by modifying a few configuration constants the two implementations share, we were able to compile them and measure RAM utilization and code size. Table 4 shows the results. The fiber's stack accounts for 512 bytes of the native runtime RAM overhead.

An ASVM doubles the size of the TinyOS image, but this is a one time cost for a wide range of regions programs. Reprogramming the native implementation requires sending a total of nineteen kilobytes: reprogramming the RegionsVM implementation requires sending a seventy byte ASVM handler, less than 0.5% of the size of the binary. Additionally, handlers run in the sandboxed virtual environment, and benefit from all of its safety guarantees. If, after many retaskings, the user decides that the particular networking abstractions an ASVM provides are not quite right, a new one can always be installed using binary reprogramming.

Deluge is the standard TinyOS system for disseminating binary images into a network [11]. Reported experimental results on a network similar to the one we used in our propagation experiments state that disseminating 11kB takes 10,000 transmissions: disseminating the 19kB of the native implementation would take approximately 18,000 transmissions. In contrast, from the data in Table 3, a RegionsVM program takes fewer than five hundred transmissions, less than 3% of the cost, while providing safety. The tradeoff is that programs are interpreted bytecodes instead of native code, imposing a CPU energy overhead. We evaluate this cost in Sections 4.5-4.6, using microbenchmarks and an application level comparison with TinyDB.

4.3 Flexibility: Languages

ASVMs currently support three languages, TinyScript, motlle and TinySQL queries. We discuss TinySQL in Section 4.4, when presenting QueryVM.

TinyScript is a bare-bones language that provides minimalist data abstractions and control structures. It is a BASIC-like imperative language with dynamic typing and a simple data buffer type. TinyScript does not have dynamic allocation, simplifying concurrency resource analysis. The resources accessed by a handler are the union of all resources accessed by its operations. TinyScript has a one to one mapping between handlers and capsules. Figure 5 contains sample TinyScript code and the corresponding assembly it compiles to.

Motlle (MOTe Language for Little Extensions) is a dynamically-typed, Scheme-inspired language with a C-like syntax. Figure 6 shows an example of heavily commented mottle code. The main practical difference with TinyScript is a much richer data model: motlle supports vectors, lists, strings and first-class functions. This allows significantly more complicated algorithms to be expressed within the ASVM, but the price is that accurate data analysis is no longer feasible on a mote. To preserve safety, motlle serializes thread execution by reporting to the concurrency manager that all handlers access the same shared resource. Motlle programs are transmitted in a single capsule which contains all handlers.

```
settimer0(500);       // Epoch is 50s
mhop_set_update(100); // Update tree every 100s

// Define Timer0 handler
any timer0_handler() { // 'any' is the result type
  // 'mhop_send' sends a message up the tree
  // 'encode' encodes a message
  // 'next_epoch' advances to the next epoch
  //     (snooped value may override this)
  send(encode(vector(next_epoch(), id(), parent(),
                                   temp())));
}

// Intercept and Snoop run when a node forwards
// or overhears a message.
// Intercept can modify the message (aggregation).
// Fast-forward epoch if we're behind
any snoop_handler() heard(snoop_msg());
any intercept_handler() heard(intercept_msg());
any heard(msg) {
  // decode the first 2 bytes of msg into an integer.
  vector v = decode(msg, vector(2));

  // 'snoop_epoch' advances epoch if needed
  snoop_epoch(v[0]);
}
```

Figure 6: A motlle data collection query: return node id, routing tree parent and temperature every 50s.

Motlle-based ASVMs therefore do not support incremental changes to running programs.

4.4 Flexibility: Applications

We have built two sample ASVMs, RegionsVM and QueryVM. RegionsVM, designed for vehicle tracking, presents the abstract regions programming abstraction of MPI-like reductions over shared tuple spaces. Users write programs in TinyScript, and RegionsVM includes ASVM functions for the basic regions library; we obtained the regions source code from its authors. Figure 7 shows regions pseudocode proposed by Welsh at al. [28] next to actual TinyScript code that is functionally identical (it invokes all of the same library functions). The nesC components that present the regions library as ASVM functions are approximately 400 lines of nesC code.

QueryVM is designed for periodic data collection using the aggregated collection routing abstraction mentioned in Section 3.6. QueryVM provides a TinySQL programming interface, similar to TinyDB, presenting a sensor network as a streaming database. TinySQL's main extension to SQL is a 'sample period' at which the query is repeated. TinySQL supports both simple data collection and aggregate queries such as

SELECT AVG(temperature) INTERVAL 50s

to measure the average temperature of the network. The latter allow in-network processing to reduce the amount of traffic sent, by aggregating as nodes route data [20].

In our implementation, TinySQL compiles to motlle code for the handlers that the aggregation collection tree library provides. This has the nice property that, in ad-

```
location = get_location();
/* Get 8 nearest neighbors */
region = k_nearest_region_create(8);

while(true) {
  reading = get_sensor_reading();

  /* Store local data as shared variables */
  region.putvar(reading_key, reading);
  region.putvar(reg_x_key, reading * location.x);
  region.putvar(reg_y_key, reading * location.y);

  if (reading > threshold) {
    /* ID of the node with the max value */
    max_id = region.reduce(OP_MAXID, reading_key);

    /* If I am the leader node...  */
    if (max_id == my_id) {
      sum = region.reduce(OP_SUM, reading_key);
      sum_x = region.reduce(OP_SUM, reg_x_key);
      sum_y = region.reduce(OP_SUM, reg_y_key);
      centroid.x = sum_x / sum;
      centroid.y = sum_y / sum;
      send_to_basestation(centroid);
    }
  }
  sleep(periodic_delay);
}
```

(a) Regions Pseudocode

```
!!  Create nearest neighbor region
KNearCreate();

for i = 1 until 0
  reading = int(mag());

  !!  Store local data as shared variables
  KNearPutVar(0, reading);
  KNearPutVar(1, reading * LocX());
  KNearPutVar(2, reading * LocY());

  if (reading > threshold) then
    !!  ID of the node with the max value
    max_id = KNearReduceMaxID(0);

    !!  If I am the leader node
    if (max_id = my_id) then
      sum = KNearReduceAdd(0);
      sum_x = KNearReduceAdd(1);
      sum_y = KNearReduceAdd(2);
      buffer[0] = sum_x / sum;
      buffer[1] = sum_y / sum;
      send(buffer);
    end if
  end if
  sleep(periodic_delay);
next i
```

(b) TinyScript Code

Figure 7: Regions Pseudocode and Corresponding TinyScript. The pseudocode is from "Programming Sensor Networks Using Abstract Regions." The TinyScript program on the right compiles to 71 bytes of binary code.

dition to TinySQL, QueryVM also supports writing new attributes and network aggregates in motlle. In contrast, TinyDB is limited to the set of attributes and aggregates compiled into its binary.

4.5 Efficiency: Microbenchmarks

Our first evaluation of ASVM efficiency is a series of microbenchmarks of the scheduler. We compare ASVMs to Maté, a hand-tuned and monolithic implementation.

Following the methodology we used in Maté [15], we measured the bytecode interpretation overhead an ASVM imposes by writing a tight loop and counting how many times it ran in five seconds on a mica mote. The loop accessed a shared variable (which involved lock checks through the concurrency manager). An ASVM can issue just under ten thousand instructions per second on a 4MHz mica, i.e., roughly 400 cycles per instruction. The ASVM decomposition imposes a 6% overhead over a similar loop in Maté, in exchange for handler and instruction set flexibility as well as race-free, deadlock-free parallelism.

We have not optimized the interpreter for CPU efficiency. The fact that high-level operations dominate program execution [15], combined with the fact that CPUs in sensor networks are generally idle, makes this overhead acceptable, although decreasing it with future work is of course desirable. For example, a KNearReduce function in the RegionsVM sends just under forty packets, and its ASVM scripting overhead is approximately 600 CPU cycles, the energy overhead is less than 0.03%. However, a cost of 400 cycles per bytecode means that implementing complex mathematical codes in an ASVM is inefficient; if an application domain needs significant processing, it should include appropriate operations.

	None	Operation	Script
Iteration	16.0	16.4	62.2
Sort Time	-	0.4	46.2

Table 5: Execution time of three scripts, in milliseconds. None is the version that did not sort, operation is the version that used an operation, while script is the version that sorted in script code.

To obtain some insight into the tradeoff between including functions and writing operations in script code, we wrote three scripts. The first script is a loop that fills an array with sensor readings. The second script fills the array with sensor readings and sorts the array with an operation (`bufsorta`, which is an insertion sort). The third script also insertion sorts the array, but does so in TinyScript, rather than using an operation. To measure the execution time of each script, we placed it in a 5000 iteration loop and sent a UART packet at script start and end. Table 5 shows the results. Sorting the array with script code takes 115 times as long as sorting with an operation, and dominates script execution time. Interpretation is inefficient, but pushing common and expen-

```
// Initialise the operator
expdecay_make = fn (bits) vector(bits, 0);
// Update the operator (s is result from make)
expdecay_get = fn (s, val)
  // Update and return the average (s[0] is BITS)
  s[1] = s[1] - (s[1] >> s[0]) + (attr() >> s[0]);
```

Figure 8: An exponentially decaying average operator for TinySQL, in motlle.

Name	TinySQL
Simple	`SELECT id,parent,temp INTERVAL 50s`
Conditional	`SELECT id, expdecay(humidity, 3) WHERE parent > 0 INTERVAL 50s`
SpatialAvg	`SELECT AVG(temp) INTERVAL 50s`

Table 6: The three queries used to evaluate data collection implementations. TinyDB does not directly support time-based aggregates such as `expdecay`, so in TinyDB we omit the aggregate.

sive operations into native code with functions minimizes the amount of interpretation. Section 4.7 shows that this flexible boundary, combined with the very low duty cycle common to sensor networks, leads to interpretation overhead being a negligible component of energy consumption for a wide range of applications.

4.6 Efficiency: Application

QueryVM is a motlle-based ASVM designed to support the execution of TinySQL data collection queries. Our TinySQL compiler generates motlle code from queries such as those shown in Table 6; the generated code is responsible for timing, data collection message layout and how to process or aggregate data on each hop up the routing tree. The code in Figure 6 is essentially the same as that generated for the Simple query. Users can write new attributes or operators for TinySQL using snippets of motlle code. For instance, Figure 8 shows two lines of motlle code to add an exponentially-decaying average operator, which an example in Table 6 uses.

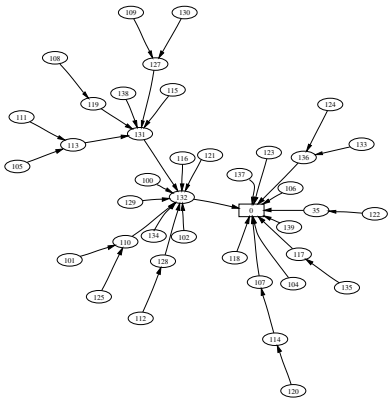


Figure 9: Tree topology used in QueryVM/TinyDB experiments. The square node is the tree root.

	Size (bytes)		Energy (mW)		Yield	
Query	TinyDB	VM	TinyDB	VM	TinyDB	VM
Simple	93	105	5.6	4.5	73%	74%
Conditional	124	167	4.2	4.0	65%	79%
SpatialAvg	62	127	3.3	3.1	46%	55%

Table 7: Query size, power consumption and yield in TinyDB and QueryVM. Yield is the percentage of expected results received.

TinySQL query results abstract the notion of time into an *epoch*. Epoch numbers are a logical time scheme that are included in query results and help support aggregation. QueryVM includes functions and handlers to support multi-hop communication, epoch handling and aggregation. QueryVM programs can use the same tree based collection layer, MintRoute [29], that TinyDB uses. QueryVM includes epoch-handling primitives to avoid replicating epoch-handling logic in every program (see usage in Figure 6). Temporal or spatial (across nodes) averaging logic can readily be expressed in motlle, but including common aggregates in QueryVM reduces program size and increases execution efficiency.

We evaluate QueryVM's efficiency by comparing its power draw to the TinyDB system on the three queries shown in Table 6. To reflect the power draw of a real deployment, we enabled low-power listening in both implementations. In low data rate networks — such as periodic data collection — low power listening can greatly improve network lifetime [22, 26]. At this level of utilization, packet length becomes an important determinant of energy consumption, so we matched the size of routing control packets between QueryVM and TinyDB. However, TinyDB's query result packets are still approximately 20 bytes longer than QueryVM's. On mica2 motes, this means that TinyDB will spend an extra 350μJ for each packet received, and 625μJ for each packet sent.

We ran the queries on a network of 40 mica2 motes spread across the ceiling of an office building. Motes had the mts400 weather board from Crossbow Technologies. Environmental changes can dynamically alter ad-hoc routing trees (e.g., choosing a 98% link over a 96% link), changing the forwarding pattern and greatly affecting energy consumption. These sorts of changes make experimental repeatability and fair comparisons unfeasible. Therefore, we used a static, stable tree in our experiments, to provide an even basis for comparison across the implementations. We obtained this tree by running the routing algorithm for a few hours, extracting the parent sets, then explicitly setting node parents to this topology, shown in Figure 9. Experiments run on adaptive trees were consistent with the results presented below.

We measured the power consumption of a mote with a single child, physically close to the root of the multi-hop network. Its power reflects a mote that overhears a lot of traffic but which sends relatively few messages (a

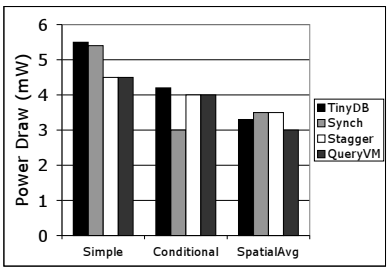


Figure 10: Power consumption of TinyDB, QueryVM, and nesC implementations. Synch is the nesC implementation when nodes start at the same time. Stagger is when the nodes start times are staggered.

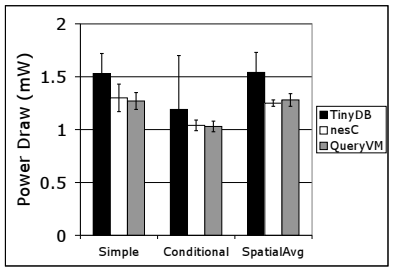


Figure 11: Average power draw measurements in a two node network. For the Conditional query, the monitored node has parent = 0, so sends no packets. The error bars are the standard deviation of the per-interval samples.

common case). In each of the queries, a node sends a data packet every 50 seconds, and the routing protocol sends a route update packet every two epochs (100 seconds). We measured the average power draw of the instrumented node over 16 intervals of 100 seconds, sampling at 100 Hz (10,000 instantaneous samples).

Table 7 presents the results from these experiments. For the three sample queries, QueryVM consumes 5% to 20% less energy than TinyDB. However, we do not believe all of this improvement to be fundamental to the two approaches. The differences in yield mean that the measured mote is overhearing different numbers of messages — this increases QueryVM's power draw. Conversely, having larger packets increases TinyDB's power draw — based on the 325μJ per-packet cost, we estimate a cost of 0.2–0.5mW depending on the query and how well the measured mote hears more distant motes.

However, these are not the only factors at work, as shown by experiments with a native TinyOS implementation of the three queries. We ran these native implementations in two scenarios. In the first scenario, we booted all of the nodes at the same time, so their operation was closely synchronized. In the second, we staggered node boots over the fifty second sampling interval. Figure 10 shows the power draw for these two scenarios, alongside that of TinyDB and QueryVM. In the synchronized case, yields for the native implementations varied between 65% and 74%, in the staggered case, yields were between 90% and 97%. As these results show, details of the timing of transmissions have major effects on yield and power consumption. To separate these networking effects from basic system performance, Section 4.7 repeats our experiments in a two-node network.

4.7 Efficiency: Interpretation

In our two-node experiments, the measured mote executes the query and sends results, and the second mote is a passive base station. As the measured node does not forward any packets or contend with other transmitters, its energy consumption is the cost of query execution and reporting. The extra cost of sending TinyDB's larger result packets is negligible (.01mW extra average power draw). We ran these experiments longer than the full network ones: rather than 16 intervals of length 100 seconds (25 minutes), we measured for 128 intervals (3.5 hours).

The results, presented in Figure 11, show that QueryVM has a 5–20% energy performance improvement over TinyDB. Even though an ASVM based on reusable software components and a common template, rather than a hand-coded, vertically integrated system, QueryVM imposes less of an energy burden on a deployment. In practice though, power draw in a real network is dominated by networking costs — QueryVM's 0.25mW advantage in Figure 11 would give at most 8% longer lifetime based on the power draws of Figure 10.

To determine where QueryVM's power goes, we compared it to four hand coded TinyOS programs. The first program did not process a query: it just listened for messages and handled system timers. This allows us to distinguish the cost of executing a query from the underlying cost of the system. The other three were the nesC implementations of the queries used for Figure 10. They allow us to distinguish the cost of executing a query itself from the overhead an ASVM runtime imposes. The basic system cost was 0.76 mW. Figure 11 shows the comparison between QueryVM and a hand-coded nesC implementation of the query. The queries cost 0.28–0.54 mW, and the cost of the ASVM is negligible.

This negligible cost is not surprising: for instance, for the conditional query, QueryVM executes 49 instructions per sample period, which will consume approximately 5ms of CPU time. Even on a mica2 node, whose CPU power draw is a whopping 33 mW due to an external oscillator (other platforms draw 3–8 mW), this corresponds to an average power cost of 3.3μW. In the 40 node network, the cost of snooping on other node's results will increase power draw by another 20μW. Finally, QueryVM sends viral code maintenance messages every 100 min-

utes (in steady state), corresponding to an average power draw of 1.6μW.

From the results in Table 7, with a power consumption of 4.5mW, a pair of AA batteries (2700mAh, of which approximately two thirds is usable by a mote) would last for 50 days. By lowering the sample rate (every fifty seconds is a reasonably high rate) and other optimizations, we believe that lifetimes of three months or more are readily achievable. Additionally, the energy cost of ASVM interpretation is a negligible portion of the whole system energy budget. This suggests that ASVM-based active sensor networking can be a realistic option for long term, low-duty-cycle data collection deployments.

5. RELATED WORK

The Maté virtual machine [15] forms the basis of the ASVM architecture. ASVMs address three of Maté's main limitations: flexibility, concurrency, and propagation. SensorWare [5] is another proposal for programming nodes using an interpreter: it proposes using Tcl scripts. For the devices SensorWare is designed for — iPAQs with megabytes of RAM — the verbose program representation and on-node Tcl interpreter can be acceptable overheads: on a mote, however, they are not.

SOS is a sensor network operating system that supports dynamic native code updates through a loadable module system [8]. This allows small and incremental binary updates, but requires levels of function call indirection. SOS therefore sits between the extremes of TinyOS and ASVMs, where its propagation cost is less than TinyOS and greater than ASVMs, and its execution overhead is greater than TinyOS but less than ASVMs. By using native code to achieve this middle ground, SOS cannot provide all of the safety guarantees that an ASVM can. Still, the SOS approach suggests ways in which ASVMs could dynamically install new functions.

The Impala middleware system, like SOS, allows users to dynamically install native code modules [18]. However, unlike SOS, which allows modules to both call a kernel and invoke each other, Impala limits modules to the kernel interfaces. Like ASVMs, these interfaces are event driven, and bear a degree of similarity to Maté. Unlike ASVMs, however, Impala does not provide general mechanisms to change its triggering events, as it is designed for a particular application domain, ZebraNet [13].

Customizable and extensible abstraction boundaries, such as those ASVMs provide, have a long history in operating systems research. Systems such as scheduler activations [3] show that allowing applications to cooperate with a runtime through rich boundaries can greatly improve application performance. Operating systems such as exokernel [14] and SPIN [4] take a more aggressive approach, allowing users to write the interface and improve performance through increased control. In sensor networks, performance — the general goal of *more*, whether it be bandwidth, or operations per second — is rarely a primary metric, as low duty cycles make resources plentiful. Instead, robustness and energy efficiency are the important metrics.

ANTS, PLAN, and Smart Packets are example systems that bring active networking to the Internet. Although all of them made networks dynamically programmable, each system had different goals and research foci. ANTS focuses on deploying protocols in a network, PLANet explores dealing with security issues through language design [10, 9], and Smart Packets proposes active networking as a management tool [24]. ANTS uses Java, while PLANet and Smart Packets use custom languages (PLAN and Sprocket, respectively). Based on an Internet communication and resource model, many of the design decisions these systems made (e.g., using a JVM) are unsurprisingly not well suited to mote networks. One distinguishing characteristic in sensor networks is their lack of strong boundaries between communication, sensing, and computation. Unlike in the Internet, where data generation is mostly the province of end points, in sensor networks every node is both a router and a data source.

Initial mote deployment experiences have demonstrated the need for simple network programming models, at a higher level of abstraction than per-node TinyOS code. This has led to a variety of proposals, including TinyDB's SQL queries [19], diffusion's aggregation [12], regions' MPI-like reductions [28], or market based macroprogramming's pricings [21]. Rather than define a programming model, ASVMs provide a way to implement and build the runtime underlying whichever model a user needs.

6. DISCUSSION AND FUTURE WORK

Section 4 showed that an ASVM is an effective way to efficiently provide a high-level programming abstraction to users. It is by no means the only way, however. There are two other obvious approaches: using a standard virtual machine, such as Java, and sending very lightweight native programs.

As a language, Java may be a suitable way to program a sensor network, although we believe a very efficient implementation might require simplifying or removing some features, such as reflection. Java Card has taken such an approach, essentially designing an ASVM for smart cards that supports a limited subset of Java and different program file formats. Although Java Card supports a single application domain, it does provide guidance on how an ASVM could support a Java-like language.

Native code is another possible solution: instead of being bytecode-based, programs could be native code stringing together a series of library calls. As sensor mote CPUs are usually idle, the benefit native code provides — more efficient CPU utilization — is minimal,

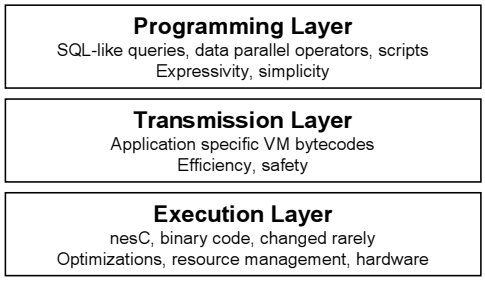


Figure 12: A layered decomposition of in-situ reprogramming.

unless a user wants to write complex mathematical codes. In the ASVM model, these codes should be written in nesC, and exposed to scripts as functions. Additionally, native code poses many complexities and difficulties, which greatly outweigh this minimal benefit, including safety, conciseness, and platform dependence. However, the SOS operating system suggests ways in which ASVMs could support dynamic addition of new functions.

ASVMs share the same high-level goal as active networking: dynamic control of in-network processing. The sort of processing proposed by systems such as ANTS and PLANet, however, is very different than that which we see in sensor nets. Although routing nodes in an active Internet can process data, edge systems are still predominantly responsible for generating that data. Correspondingly, much of active networking focused on protocol deployment. In contrast, motes simultaneously play the role of both a router and a data generator. Instead of providing a service to edge applications, active sensor nodes are the application.

Section 4.6 showed how an ASVM — QueryVM — can simultaneously support both SQL-like queries and motlle programs, compiling both to a shared instruction set. In addition to being more energy efficient than a similar TinyDB system, QueryVM is more flexible. Similarly, RegionsVM has several benefits — code size, concurrency, and safety — over the native regions implementation.

We believe these advantages are a direct result of how ASVMs decompose programming into three distinct layers, shown in Figure 12. The highest layer is the code is a user writes (e.g., TinyScript, SQL). The middle layer is the active networks representation the program takes as it propagates (ASVM bytecodes). The final layer is the representation the program takes when it executes on a mote (an ASVM).

TinyDB combines the top two layers: its programs are binary encodings of an SQL query. This forces a mote to parse and interpret the query, and determine what actions to take on all of the different events coming into the system. It trades off flexibility and execution efficiency for propagation efficiency. Separating the programming layer and transmission layer, as QueryVM does, leads to greater program flexibility and more efficient execution.

Regions combines the bottom two layers: its programs are TinyOS images. Using the TinyOS concurrency model, rather than a virtual one, limits the native regions implementation to a single thread. Additionally, even though its programs are only a few lines long — compiling to seventy bytes in RegionsVM — compiling to a TinyOS image makes its programs tens of kilobytes long, trading off propagation efficiency and safety for execution efficiency. Separating the transmission layer from the execution layer, as RegionsVM does, allows high-level abstractions to minimize execution overhead and provides safety.

7. CONCLUSION

The constrained application domains of sensor networks mean that programs can be represented as short, high level scripts. These scripts control — within the protocols and abstractions a domain requires — when motes generate data and what in-network processing they perform. Vision papers and existing proposals for sensor network programming indicate that this approach will not the exception in these systems but the rule. Pushing processing as close to the data sources as possible transforms a sensor network into an active sensor network. But, as sensor networks are so specialized, the exact form active sensor networking takes is an open question, a question that does not have a single answer.

Rather than propose a particular active networking system, useful in some circumstances and not in others, we have proposed using application specific virtual machines to easily make a sensor network active, and described an architecture for building them. Two sample VMs, for very different applications and programming models, show the architecture to be flexible and efficient. This efficiency stems from the flexibility of the virtual/native boundary, which allows programs to be very concise. Conciseness reduces interpretation overhead as well as the cost of installing new programs. "Programming motes is hard" is a common claim in the sensor network community; perhaps we have just been programming to the wrong interface?

Acknowledgements

This work was supported, in part, by the Defense Department Advanced Research Projects Agency (grants F33615-01-C-1895 and N6601-99-2-8913), the National Science Foundation (grants No. 0122599 and NSF IIS-033017), California MICRO program, and Intel Corporation. Research infrastructure was provided by the National Science Foundation (grant EIA-9802069).

8. REFERENCES

[1] A. Acharya, M. Uysal, and J. Saltz. Active disks: programming model, algorithms and evaluation. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 81–91. ACM Press, 1998.

[2] E. Amir, S. McCanne, and R. Katz. An active service framework and its application to real-time multimedia transcoding. In *SIGCOMM '98: Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 178–189. ACM Press, 1998.

[3] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[4] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15)*, 1995.

[5] A. Boulis, C.-C. Han, and M. B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services (MobiSys 2003)*, 2003.

[6] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, June 2003.

[7] L. Girod, T. Stathopoulos, N. Ramanathan, J. Elson, D. Estrin, E. Osterweil, and T. Schoellhammer. A system for simulation, emulation, and deployment of heterogeneous sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 201–213. ACM Press, 2004.

[8] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A dynamic operating system for sensor nodes. In *MobiSYS '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, 2005.

[9] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. Plan: A packet language for active networks. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 1998.

[10] M. Hicks, J. T. Moore, D. S. Alexander, C. A. Gunter, and S. Nettles. Planet: An active internetwork. In *Proceedings of IEEE INFOCOM*, 1999.

[11] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the Second International Conferences on Embedded Network Sensor Systems (SenSys)*, 2004.

[12] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proceedings of the International Conference on Mobile Computing and Networking*, Aug. 2000.

[13] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. Peh, and D. Rubenstein. Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with zebranet. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, oct 2002.

[14] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on Exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, October 1997.

[15] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, Oct. 2002.

[16] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Simulating large wireless sensor networks of tinyos motes. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, 2003.

[17] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code maintenance and propagation in wireless sensor networks. In *First USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, 2004.

[18] T. Liu and M. Martonosi. Impala: a middleware system for managing autonomic, parallel sensor systems. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 107–118. ACM Press, 2003.

[19] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: An acquisitional query processing system for sensor networks. *Transactions on Database Systems (TODS)*, 2005.

[20] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proceedings of the ACM Symposium on Operating System Design and Implementation (OSDI)*, Dec. 2002.

[21] G. Mainland, L. Kang, S. Lahaie, D. Parkes, and M. Welsh. Using virtual markets to program global behavior in sensor networks. In *Proceedings of the 11th ACM SIGOPS European Workshop*, Leuven, Belgium, 2004.

[22] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *Proceedings of the Second ACM Conferences on Embedded Networked Sensor Systems (SenSys)*, 2004.

[23] K. Römer, C. Frank, P. J. Marrón, and C. Becker. Generic role assignment for wireless sensor networks. In *Proceedings of the 11th ACM SIGOPS European Workshop*, Leuven, Belgium, 2004.

[24] B. Schwartz, A. W. Jackson, W. T. Strayer, W. Zhou, R. D. Rockwell, and C. Partridge. Smart packets: Applying active networks to network management. *ACM Transations on Computer Systems*, 2000.

[25] C. Sharp, S. Shaffert, A. Woo, N. Sastry, C. Karlof, S. Sastry, and D. Culler. Design and implementation of a sensor network system for vehicle tracking and autonomous interception. Proceedings of the Second European Workshop of Wireless Sensor Networks (EWSN 2005), 2005.

[26] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. An analysis of a large scale habitat monitoring application. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys 2004)*, 2004.

[27] D. Tennenhouse and D. Wetherall. Towards an active network architecture. In *Computer Communication Review, 26(2)*, 1996.

[28] M. Welsh and G. Mainland. Programming sensor networks with abstract regions. In *First USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, 2004.

[29] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of the first international conference on Embedded networked sensor systems*, pages 14–27. ACM Press, 2003.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

Member Benefits

- Free subscription to *;login:*, the Association's magazine, published six times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *;login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see *http://www.usenix.org/membership/specialdisc.html* for details.

SAGE

SAGE is a Special Interest Group (SIG) of the USENIX Association. It is organized to advance the status of computer system administration as a profession, establish standards of professional excellence and recognize those who attain them, develop guidelines for improving the technical and managerial capabilities of members of the profession, and promote activities that advance the state of the art or the community.

USENIX & SAGE Thank Their Supporting Members

USENIX Supporting Members

❖ Addison-Wesley/Prentice Hall PTR ❖ Ajava Systems, Inc. ❖ AMD ❖ Asian Development Bank ❖
❖ Atos Origin BV ❖ Cambridge Computer Services, Inc. ❖ Delmar Learning ❖
❖ DoCoMo Communications Laboratories USA, Inc. ❖ Electronic Frontier Foundation ❖
❖ Hewlett-Packard ❖ IBM ❖ Intel ❖ Interhack ❖ MacConnection ❖
❖ The Measurement Factory ❖ Microsoft Research ❖ Oracle ❖ OSDL ❖ Perfect Order ❖
❖ Portlock Software ❖ Raytheon ❖ Sun Microsystems, Inc. ❖ Taos ❖ Tellme Networks ❖
❖ UUNET Technologies, Inc. ❖ Veritas Software ❖

SAGE Supporting Members

❖ Addison-Wesley/Prentice Hall PTR ❖ Ajava Systems, Inc. ❖ Asian Development Bank ❖
❖ Fotosearch ❖ Microsoft Research ❖ MSB Associates ❖ Raytheon ❖
❖ Ripe NCC ❖ Taos ❖ Tellme Networks ❖

For more information about membership, conferences, or publications,
see *http://www.usenix.org/*
or contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 Fax: 510-548-5738 Email: *office@usenix.org*

ISBN 1-931971-30-7